AD-A084 195 MARTIN MARIETTA AEROSPACE DENVER CO DENVER DIV F/6 9/2
RECOMMENDATIONS FOR A RETARGETABLE COMPILER,(U)
MAR 80 G E HEYLIGER, L L MCELHANEY, T H DWYER F30602-78-C-0297
UNCLASSIFIED MCR-79-631 RADC-TR-79351 NL

RADC-TR-79-351
Final Technical Report
March 1980

(12)

LEVEL II

RECOMMENDATIONS FOR A RETARGETABLE COMPILER

Martin Marietta Aerospace

George E. Heyliger
Lyle L. McElhaney
Timothy H. Dwyer
Patrick J. Keziah

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RECOMMENDATIONS FOR A RETARGETABLE COMPLIER

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-351 has been reviewed and is approved for publication.

APPROVED: Samuel A. Di Nitto

SAMUEL A. DI NITTO, JR.
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-79-351	2. GOVT ACCESSION NO.: AD-A084293
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): RECOMMENDATIONS FOR A RETARGETABLE COMPILER	5. TYPE OF REPORT & PERIOD COVERED: Final [redacted] Report 20 Sep 78 - 19 Sep 79
	6. PERFORMING ORG. REPORT NUMBER: MCR-79-631
7. AUTHOR(s): George E. Heyliger, Timothy H. Dwyer, Lyle L. McElhaney, Patrick J. Keziah	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0297
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Martin Marietta Aerospace, P O Box 179, Denver CO 80102	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F 5581[redacted]
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: Mar[redacted]80
	13. NUMBER OF PAGES: 149
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Samuel A. Di Nitto, Jr. (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Compiler	Computer Description Languages
Code Generator	J73
Retargetable Compiler	Ada
Intermediate Language	Programming Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report summarizes the results of a study into the design of compilers that are more easily retargetted than compilers of today. Prime emphasis is on the use of machine description language to define the "target" machine, and a common intermediate language to "target to." The goal was to define the characteristics for a tool which would take the description of a computer as input and output a code generator for that machine.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SUMMARY

Software portability is one of the key methods of combating the software crisis brought on by plummetting hardware costs and proliferation of new computer designs. Use of High Order Languages (HOL's) assist this effort, as long as the corresponding compilers can be easily generated to match a given HOL to any new machine architecture. The Retargetable Compiler is such a compiler, in which an HOL program may be reduced to high quality code for a given machine. The compiler should be automatically produced from a formal description of the machine.

In this report we will review the current level of technology covering compiler theory, and especially compiler-compiler theory. We will establish bounds on the kinds of languages and machines we intend to be able to handle with our design, and finally we will present a design for a retargetable compiler system based on our research performed under this contract.

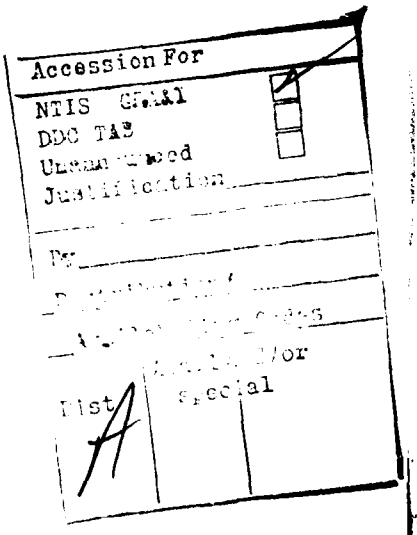


Table of Contents

Appendices

Figures

EVALUATION

The objective of this effort was to develop design recommendations for a tool to automatically build the code generator portion of a compiler (automatically retarget) from a formal description of the target machine. Such a tool would enable the use of manpower efficient high order computer programming languages on more Air Force software system developments. Since software development and maintenance is labor intensive, this effort is obviously responsive to RADC TPO-R5A, "Software Cost Reduction".

This effort met the objective with a thoroughly analyzed set of recommendations for such a tool. The present plan is to develop a "retargetable" compiler for the new common DOD programming language known as Ada. It is hoped that the retargetable compiler for Ada can commence development in FY82.

Samuel A. Di Nitto

SAMUEL A. DI NITTO, JR
Project Engineer

1. INTRODUCTION

In this section, we discuss the current level of compiler technology, including its strengths and weaknesses, as well as our goals under the Retargetable Compiler project.

1.1 The Current State of Compiler Technology

Since the introduction of high order languages, both the theory and practice of language compilation have been advanced. One of the more significant factors in this advancement is the use of meta-languages to describe the syntax of the target language. Two beneficial effects have resulted. First, the front-end parts of compilers, which decompose the lexical/syntactic structure of the source language, have been generalized to the point that they can operate directly from the formal specification of the source language. Thus, source decomposition is no longer considered a major hurdle in compiler development. The second benefit of the use of meta-languages is simplification of programming language through a tendency to use fewer and more systematic syntactic structures. While this has only a quantitative effect on compilers, it is a significant advancement in practical software development.

Success with formalization of language syntax has led to the desire for some formalizations of semantics and program flow. Making use of these formalizations, the compilation process has been evolved into a two step process, in which the source program is first translated into an intermediate language, and then into the target language.

The intermediate language is designed to express only the semantics of the program, thereby isolating the syntax of the source language from that of the target language. There is a large variety of global and local optimizations which are most conveniently performed on the intermediate language form of a program. These optimizations (which

should more properly be called "improvements") are transformations of the program which do not alter the semantics of the program as specified by its source language form, and which reduce some compile time measure of program cost. The cost function to be reduced is usually program size, estimated program speed, or a combination. The optimizations most easily performed on the intermediate language program (before code generation) are often called "machine independent optimizations". They can be performed regardless of the target machine, although their desirability may depend on certain aspects of the target machine. There is a large body of theory on obtaining and using the data flow information on which global optimization is based, and a less comprehensive though growing body of experience in using machine independent optimizations.

1.2 Problem Areas in Compiler Technology

After a source program has undergone semantic analysis and machine independent optimization, it must be translated into the target language. This translation is from an intermediate language, which expresses the program being compiled. The most common approach is to generate target code based upon the elements of the intermediate language, with consideration given to the context of each such element. This approach, termed the "ad hoc" approach, is basically a heuristic approach as opposed to one based strictly on a theory of code generation. There is more theory known concerning register allocation strategies, but up to this point the theory is applicable only for special cases, such as single register machines or single code blocks without loops. The practical algorithm for register allocation are again heuristics. This lack of underlying theory is a major obstacle to automatic generation of the register allocation and code generation phases of a compiler.

Another area in which there is plenty of information and little organization is target dependent optimization. This includes all

transformations applied to either intermediate or target forms which can be shown to constitute "improvement" of the program only when characteristics of the target are given. Most of these optimizations are obvious, and of direct benefit in reducing target program size or execution time, but they all seem to be unique in comparison with each other. They do not form categories or patterns within a coherent theory. Each is its own category, and each is either completely applicable or completely in-applicable for each target machine. If a fully generalized theory of code generation existed, it would have to be sufficiently adaptive to the target architecture that it would probably eliminate the need for our present distinction between code generation, register allocation, and target dependent optimization.

1.3 Other Compiler Development Problems

In developing any software project, it is valuable to have not only a clean set of objectives, but also a way of measuring progress toward those objectives. A compiler may well be an extreme case in this regard. Compilers possess many characteristics, and meeting an objective relative to one characteristic often conflicts with others. In the course of developing a compiler, these conflicts must be worked out through compromise. Such compromises can be made rationally only if the developer knows the degree to which he is trading one thing for another. Hence, it is especially important to have techniques for measuring progress toward objectives, and for analyzing the compromise between them. For example, consider two typical objectives: to minimize target program size(s), and to minimize target program execution time(t). It is simple enough to choose between two optimization techniques if the difference between those techniques affects only s or only t, but this is rarely the case in light of potential subsequent optimizations. We need an algorithm to trade between s and t. Perhaps it is to minimize (s * t), (As + t), or ($s^I + t^J$). The algorithm itself may not matter as much as the fact that it may change from one segment of the program to another (e.g., local minimization of t has

more affect on global t if the locale is in a loop). If we define optimization as a tendency toward less comsumption of a single resource (or fixed proportion of several resources), then we can optimize only if we can determine the consumption of that single resource. In fact, most optimization decisions can be made based on a simple resource (s, or t) and within a limited context. But without a philosophy for resource definition and comparative measure, we will not have a consistent and justifiable approach to the more complex optimizations.

When measuring the quality of a compiler some attention might be given to the cost of compilation. The compile-time resources that should be spent for a given improvement in object program quality depend on the intended use of the compiler. For the compiler writer to determine the best optimization and code generation strategies to apply, he will need to be able to make estimates of the cost/benefit ratios of the alternate techniques.

1.4 The Retargetable Compiler Goals

Much of theory and practice of compilation and compiler generation has been developed from the bottom up. That is, only specific problems have been addressed, and these only in fairly narrow contexts. This is not to discount the value of the work already done. We would like to place this existing knowledge in a framework of needs and requirements that is derived from the top down. This would clarify the gaps in our knowledge, and the consequent objective of closing those gaps. Additionally, it would help us bring some sense and uniformity to this fragmented field, and could lead to a philosophical foundation which is necessary for rapid progress.

The whole field of automatic compiler generation is not yet mature, and is not likely to become so for some time. We can, however, make use of what we do know provided that we are practical. Toward this end, we intend to concentrate in areas of theory and technology which

appear to have the greater pay-offs in features that are desirable in a compiler generator. Those areas which have minimal pay-off, or which cannot be implemented without a great expense will be addressed only in regard to possible future work. In this way, we will provide the basis for a practical retargetable compiler generator which exhibits good balance between retargeting effort and target program quality.

2. SCOPE

In this section we will deal with those constraints which we feel realistically limit the goals of the retargetable compiler, and thereby define the goals of this work.

2.1 Languages of Interest

The programming languages for which retargetability would be useful are legion, and more are developed every day. Our approach to optimization and code generation issues are best served by limiting the languages to those which handle the data structures for which the general purpose machines of today are designed: integer and real arithmetic, and composites of these. Some other data types, which are to some extent machine supported, are either not universal enough (e.g., decimal arithmetic) or not uniformly treated (e.g., byte strings) across the range of computers of interest. The languages included are those known as either "algebraic" languages, or "system programming" languages, and include FORTRAN, BASIC, ALGOL, PASCAL, BLISS, and, in particular JOVIAL and Ada. We exclude those languages which require interpretation or extreme run-time library support, such as SNOBOL, LISP, and APL. These languages are generally characterized by their support of some specific data structure such as strings or lists.

The reasons for this emphasis on languages which are machine data-structure supported are that they are the ones for which extensive optimization strategies and clever code generation procedures will provide the best improvements in the finished object code. Those languages which are heavily run-time supported (in order to handle data structures divorced from existing machine capabilities) are not so readily optimal - most such work is performed in writing the run-time support package for each machine.

Note, however, that it will probably be impossible to build any compiler for any machine without some kind of run-time support available, particularly in the areas of input/output processing, storage allocation, and other operating system interfaces.

2.2 Target Machines

The choice of target machine for the retargetable compiler must likewise be limited. We must (at least for this report) discard the special purpose machines such as array processors, and those specialized for high level languages, such as pure stack machines. We will not be considering machines which are designed to specifically implement parallel processes, except insofar as they will perform as a single processor on the target code.

Note that these exclusions do not seriously limit the range of general purpose processors in use today. We do plan to be able to generate code for one address(Intel 8080), two address (PDP-11), three address (VAX) and general register architectures (S/370, 1108) as well as combinations of these sets (MODCOMP). This will include most of the micro-processors available today, as well as most minis and mainframes, and even the classes of micro-code known as "vertical".

The primary reasons for the restrictions on the machine applicability is the lack of unifying compiler theory across all the more special purpose machines. The somewhat tenuous existing theory of machine-independent optimization, for example, does not at all address the equivalent problem in truly parallel processes.

Finally, the retargetable compiler is envisioned to be just that - a compiler not an interpreter. We will not discuss such techniques as run-time monitoring (in the sense of Knuth's [Knu74] use of the term) and dynamic optimization, since these techniques are not yet practically useable. They are indeed subjects for future research, and future design of the Retargetable Compiler should allow for the possibilities of such techniques.

3. THE PROBLEM

3.1 Interpretation of the SOW

The Statement of Work [RFP] for the Retargetable Compiler contract is broken into two parts:

- investigation of ways and means of automatic code generation, and
- "detailed recommendations for the design" of a retargetable compiler.

The first section contains three primary areas of research:

- the CDL to drive the code generator must be specified/developed;
- a complete set of optimizations, driven by the CDL description, must be specified; and
- well specified intermediate representation must be developed.

The last section of the SOW specifies that the design must include implementations for J73/I and Ada.

We will present, in section 4.0 of this report, both our design and the algorithms we feel fill the requirements of a multi-machine environment. In this section, we will specifically address several of the issues covered by our research into Retargetable Compiler methods, and address the J73/I and Ada requirements in particular.

The SOW assumes that compiler front end technology is well enough advanced that there is no need to go into that part of the retargetable compiler. As far as the research portion of the SOW is concerned, this is a correct assumption. However, we feel the need to discuss the front end processing in the system design for four reasons:

- The requirement that we specify the intermediate language (which is input to code generation and output from the compiler front-end) infers that the front-end must supply said language. Unless we choose some already existing front-end and its intermediate language or equivalent, then we must at least hypothesize a front-end meeting this requirement.
- Some optimizations of the machine-independent, flow analysis type require data concerning the semantics of the language being translated, the generation of which is not among the capabilities of currently existing compiler-compilers. This will be considered in more detail when discussing the intermediate language.
- The final paragraph in the SOW requires the design to be applicable to (at least) two different languages. This requirement is most readily compiled with by designing a language definition driven compiler-compiler front-end.
- Finally, from a systems engineering point of view it is best to design the entire system, both front-end and back-end, together, in order to avoid interface problems and promote a uniform approach to the entire retargetable compiler problem.

With these points in mind, we will treat the design of a complete compiler-generator rather than just a code-generator generator.

3.2 Research Topics

There are five primary areas of research connected with the retargetable compiler:

- computer description languages;
- intermediate languages;
- machine independent optimization;
- code generation techniques;
- machine dependent optimizations.

3.2.1 Computer Description Languages

Current code generation techniques are oriented toward an instruction set processor. Code generation is seen as a process of producing machine language instructions which are interpreted by some target machine. This interpretation cycle assumes the execution of one instruction at a time, with the instructions being retrieved from an instruction memory, and with the ability to make the sequence of execution dependent on data values. Additionally, current code generation techniques assume a main data memory and can exploit such common features as high-speed registers, condition codes, and multi-action instructions. Instructions are assumed to be defined strictly in terms of the values read from and written into memories. These memories include main memory, registers, the program counter, and condition codes.

These characteristics of current code generation techniques point to various traits in a CDL that would be helpful for generating code for a machine described in that CDL. First, since code generation techniques assume an interpretation cycle, with the variability between machines being in the individual instructions, the description language should focus on the behavior of these individual instructions, providing tools for exact description of their behavior. The instruction descriptions should be high level, simply describing the mappings from the inputs to the outputs of the instructions. The description language should require the presence of exactly one program memory and exactly one, clearly identified program counter. Also needed are tools for the description of the various memories of the machine. To allow the code generator to make optimization choices, the description must give the costs in time and memory space of the instructions. In addition to these content requirements for the description languages, it is also desirable that the language be easy to use and of wide applicability so that machine descriptions may be already available from other applications or easy to write if no existing description exists for a machine of interest.

We have examined fourteen languages in use today which describe digital hardware to some extent. (See Appendix A) These are:

ISP and derivatives (ISP[1], ISPS)	(Instruction Set Processor)
CDL and Purdue Extended CDL	(Computer Description Language)
DDL	(Digital Design Language)
LALSD	(A Language for Automated Logic and System Design)
SMITE	(Software Machine's Implementation Tool using Emulation)
AHPL	(A Hardware Programming Language)
APL	(A Programming Language)
MOP	(Machine Operations)
CASSANDRE	
APDL	(Algorithmic Processor Description Language)
LOGAL	(Logic Algorithm Language)
LCD	(Language for Computer Design)
ERES	(Erlanger Rechner-Entwurfs-Sprache-Erlangen Computer Description Language)
ConLan	(Consensus Language)

MOP, developed by Cattell at CMU, is an obvious candidate based on these criteria. It was developed for code generation with the above criteria in mind. It assumes an interpretation cycle, which is not made explicit in the description, which executes instructions stored in a main memory. In fact, no overall control structure or algorithm is given for the machine, with control being left to the assumed interpretation cycle. The main memory is also used to store data. Each instruction is identified and described separately, with the description giving the fields of the instruction, its operation code, its costs, and its actions. The actions are described in terms of the input and output of the instruction. Tools are provided for the exact and clear description of available memories and how they are accessed by the various instructions. Clarity of the description here includes the capacity for automated analysis. Memories are described in terms of width, length, and function. Also given are the instruction fields and field values needed to use the various access modes for the various memories. At the same time, MOP is easy to use, so machine descriptions may be easily prepared.

Wide applicability of a CDL is in conflict with the rest of the above criteria, and as is almost inevitable, MOP is too specialized for wide applicability. Particularly, its assumption of an interpretation cycle and high level of description make it unsuitable for many applications. However, ISPS, a very general and widely used language can be translated into MOP in an almost completely automatic manner [Oak79]. ISPS is a derivative of ISP and is an easy to use language that can be used at many levels of detail and provides very powerful modularization constructs which are useful for both writing and analyzing descriptions. The data types, accessing mechanisms, and operators provided by ISPS closely correspond to those existing in current computers. These close correspondences make descriptions easy to write and facilitate the writing of accurate descriptions. The simple control constructs in ISPS are easy to analyze and closely correspond to those available in computer hardware. ISPS has some powerful extension mechanisms that have made it suitable for describing a large number of computers for a wide range of applications, including a great deal of work at CMU.

It should be noted that the work necessary to produce MOP from ISPS (symbolic execution, I/O assertion development, etc.) does not duplicate much of the work in using MOP for code generation, and thus the use of MOP as the immediate input to code generation does not introduce extra work, even when the only available description of a computer is in ISPS. Also, if no ISPS or MOP description of a machine is available, a MOP description may be written by hand comparatively quickly and easily.

3.2.2 Intermediate Languages

The Intermediate language is the language used to represent user's program between passes of the compiler. In particular, it is the form between the syntactical/semantic encoding and the code generation phases, and therefore at the state where most (not all!) of the language dependent features are absent, and before any machine dependencies have occurred [CAT78,HW78].

Intermediate languages (IL) originally were designed to effectively carry information between passes of a compiler with little thought given to making them general. It was soon realized that if a suitable IL could be developed, then interchangeable front ends and back ends would reduce the work of producing compilers for n machines and m languages from producing m x n compilers to m plus n compilers, a significant reduction in effort [Col74].

With this in mind, ILs were designed for their own sake, and the ideas of language/machine independence were first explored. The first such IL was developed by Mock and Steel [MOS58] and called UNCOL. Later efforts along the same line include Coleman and Wait's JANUS [Col74, WH78] and Cattell's TCOL family of ILs [Cat78, SLN79].

Several ILs are rather language dependent, but have come to prominence because of unique features or special implementation solutions. These include HALMAT (for HAL compilers) [II74], the JOCIT IL (for JOVIAL/J3) [Dun75] and OCODE (for BCPL) [Ric71]. An extended consideration of each above named IL (except UNCOL) is given in Appendix B.

Because of their language independence, JANUS and TCOL are the only ILs from Appendix B that will be considered as the IL for the Retargetable Compiler.

JANUS was designed as a universal IL. It provides a large set of operators and a flexible storage scheme. Its control and data structures are at a fairly low level, and it uses a stack for intermediate results. It is a linear language that does not depend on a symbol table and was designed to be translated by a macro processor.

TCOL was designed by Cattell as a universal IL for use with the MMM algorithm. TCOL is a class of languages, and $TCOL_{Ada}$ is a particular member developed at Carnegie-Mellon University. TCOL programs are trees, with fairly high level data and control constructs.

	TCOL	JANUS
Source Language Independence		
Control Constructs	Tied to source language	Branches and conditionals - language independent
Data Structures	Large variety; flexible	Flexible and language independent
Machine Independence	Both languages are very machine independent	
Level		
Control Constructs	High level	Low level
Data Structures	Medium level	Medium level
Operators	High level	High level
Temporary Storage	Implied by tree structure (Implicit stack)	Values are kept on the stack
Extendability	Both languages were designed to permit extension	
Suitability for Code Generation	Designed for use with MMM algorithm - very suitable	Has to be converted to tree form for MMM algorithm
Practical Experience	None	Several Experimental Compilers

Figure 3.1 TCOL and JANUS Comparison

The machine independence and extendability of the two languages are practically equivalent. Although the language independence of $TCOL_{Ada}$ is less than that of JANUS, as mentioned in the appendix this will not be a practical obstacle, since the compiler-compiler will reduce the effort needed to adapt to changes in the IL. The considerations left are the level, temporary storage and suitability for code generation of the two languages. The first two qualities are important because they affect the suitability of the languages for code generation.

JANUS is a language that is very language independent, fairly low-level and with fairly simple data types. These qualities, and its stack mechanisms were motivated by the design goal that it be suitable for translation to machine code by a macro translator. TCOL on the other hand, has high level control constructs, with high level data types and no explicit stack. This structure is motivated by the desire to make the IL program easily manipulatable by optimizers, code generators, etc. but it makes the translation to machine code more difficult.

Both JANUS and TCOL are suitable for the optimizations and code generation techniques described in this report. TCOL's higher level makes optimizations easier, however, and its tree structure is conceptually well suited to tree-matching code generation techniques. $TCOL_{Ada}$ is therefore our choice for the Retargetable Compiler IL.

3.2.3 Machine Independent Optimizations

The theory of machine independent optimizations, as stated before, has been fairly well explored and a good basis for use has been advanced to practicality, notably with the BLISS11 compiler [WJW75].

Conceptually these optimizations can be reduced to three classes of operations: code movement, constant folding, and dead code elimination. The first can be subdivided into the several situations in which moving code (and subsequent consolidation) may provide greater efficiencies. We will mention each in turn below.

Constant folding - this optimization results from the appearance of expressions involving only constants in the program tree. These expressions could have resulted from a programmer's explicit expression (PI=3.14159;PI2= 2.*PI) or implicitly, such as in subscript evaluation (REAL A(5,2); ... B=A(2,I)). The ability to do constant folding can happen at any time from first tree build through all the other code movement optimizations, as reconstruction juxtaposes constants that were previously separated.

Dead code elimination - some of the optimizations discussed below, particularly variable and constant propagation, sometimes result in code which will not be executed under any circumstances. This dead code can be detected and eliminated.

Both constant folding and dead code elimination always should be effected as soon as the proper conditions are detected, inasmuch as they always result in a smaller, more efficient program tree. The remaining methods, classed as code movements, do not always increase the efficiency for all desirable measures of efficiency. Therefore, code movements should only be performed after the cases can be measured and compared in a meaningful way.

Redundant expression elimination - this is the case of a value being calculated more than once in a code block. Generally, the result of the first calculation may be saved and used in place of the second calculation. This kind of optimization appears most frequently as the result of subscript computations.

Hoisting code motion - when a value is identically computed in each branch from a flow path, that computation may be hoisted into the common path and eliminated in the branches. The same idea allows common statements made in branches to be dropped below the point where they flow together.

Rho motion [LCH79] -This optimization attempts to assure that a computation is performed only once in a given flow instance. For example:

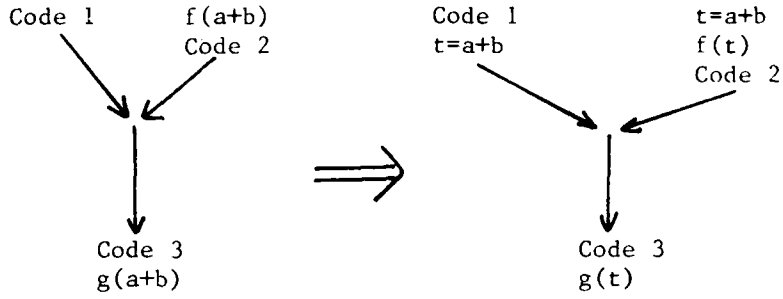


Figure 3.2 Rho Motion Optimization

This motion may not seem useful unless the common path is part of a loop. In the trivial case, this is known as moving invariant code out of loops.

Strength reduction - This optimization attempts to eliminate expensive operations through use of less expensive ones. This is particularly useful for indexing inside loops where an implicit multiplication of a subscript can be avoided through saving and later adding to the previously calculated displacement if the subscript is being increased by a constant each time around the loop.

3.2.4 Code Generation Techniques

The direct final translation into machine or assembly language for the target machine is code generation. There are several fairly well defined ways of producing this effect:

- ad hoc code generation
- macroprocessing
- code generator languages
- machine description - driven techniques

Most of the code generators behind today's compilers are written using ad hoc techniques, which means that code to handle each construct of its input is hand written as a special case on any particular machine. This is the brute force method, applied in the absence of any coherent theory

covering code generation. In order to convert the code generator for any other machine, a complete rewrite is required, with almost all of the original effort being unuseable. The efficiency of the code produced is in direct proportion to the effort expended, as Wulf has demonstrated with the BLISS 11 compiler.

Macroprocessing is a generalization of the text processing mechanisms found in many assemblers in which user generated "macros" direct the translation of text [CHM78]. They have been used extensively to provide extensions onto many high order languages, and may likewise translate some flattened intermediate structure into assembly source text. The power in the method is in direct proportion to the power of the macroprocessor used. Some processors are quite simple and easy to use; the cost is loss of flexibility. The more powerful processors allow flexibility rivalling assembly languages themselves, at the cost of difficulty in preparing the macros. In the extreme, writing the macros becomes equivalent to writing the code generator itself.

The macroprocessor approach fails because the tool itself is too general - it cannot take advantage of the reduced functional requirements that the restricted goal, code generation affords. This weakness in approach is covered by the code generator languages [DNF79]. They are, in effect, special purpose languages designed for building code generators, and thus provide the code generation flavor missing from the macroprocessors. Code generators become much easier to write than in the ad hoc method, but each one still must be written separately. Again, the quality of the code generated reflects the skill and effort expended in writing the code generator.

The final method of code generation, using a machine description, breaks from the previous techniques in that the machine dependent parts of the code generator are reduced to tables, and are generated automatically, based on a machine description [Cat78], [Fra77], [Gla73]. It is true that the description must be supplied for each machine, but that seems to be a lesser effort

than hand writing the code generator, and besides, the same description may be used in several other ways in the compiler and related projects, such as hardware design verification, diagnostics generation, etc., and thus the effort is "amortized" across a series of uses. Moreover, the machine descriptions for a number of machines already exist and could conceivably be used as is.

With all these considerations in mind, it seems that the machine description-driven methods of code generation are most appropriate for the Retargetable Compiler.

3.2.5 Machine Dependent Optimizations

This class of optimizations is not appropriately named, since several of the optimizations discussed here are largely machine independent.

The single common characteristic of these optimization techniques is that they can best be performed only after code generation is finished. Some need to know actual code sizing; some are code to code replacements (or straight eliminations) for particular code sequences.

The utter lack of theoretical basis for tying these optimizations together requires that they be treated as totally separate items. All are machine dependent in the sense that they need to know, for instance, what a branch instruction looks like, but some need no more than that, whereas some need much more data concerning the target machine.

Many such operations are available in the literature. Some of the better known and more general ones are:

cross jumping - elimination of code on two merging paths which is identical.

unreachable code - detection and elimination of code between an unconditional branch and the next label.

Branch chain elimination - if a branch instruction addresses an unconditional branch, make it a direct branch to the second target.

Redundant code elimination - elimination of instructions that are effective NOPs to the target machine.

Reversing branch sense - if a conditional branch is followed by an unconditional branch, and then the target of the first, the sense of the condition may be changed to eliminate the second branch.

Superfluous compare - delete compare instructions when previous operations have already set the condition codes.

Special case literal operands - delete instructions such as OR with literal 0, etc.

Auto increment modes - some machines can automatically increment or decrement pointers. Use of these can eliminate explicit incrementation.

Reduction to simpler form - some instructions can be reduced in size to an equivalent form, such as:

ADD #4,SP ⇒ CMP (SP)+,(SP)+

on the PDP 11.

Short form length-dependent instructions - some machines have short form instructions for use when the target operand is some small relative distance from the instruction. The algorithm to reduce these is non-trivial and requires a fair amount of analysis [Szy78].

These optimizations are only examples. No doubt new architectures will open new opportunities for novel methods of peephole optimization.

3.3 The J73/I and Ada Requirement

JOVIAL and Ada are both outgrowths of the ALGOL language. JOVIAL initially added byte and bit level constructs in order to expand ALGOL into a systems programming language, while Ada has been developed from PASCAL in order to provide PASCAL's data abstraction and modern program structuring to a language suited for use in embedded computers.

Both languages fall within the language bounds listed in the scope section of this report. Both are block oriented, procedural, algebraic languages, and both are syntacticly describable by LALR(1) grammars. They therefore will both be amenable to analysis by our automatically generated syntax analysis (see the next section). They will thus generate language independent intermediate code, suitable for code generator input.

4. A SOLUTION DESIGN

Using the studies made in the last section and our own study of the state of the art, we shall propose in this section a conceptual design for a retargetable compiler.

We will describe in general terms both the compiler generator and the compiler architecture itself. This is followed by more detailed discussions of each module in the back-end of the compiler, which is the area of primary interest to this report.

Finally, we have attached a more detailed description of our primary descriptive input, the MOP. We have included this so that the reader may more readily see the kind of description it is, and the data it contains.

4.1 Compiler Generator Architecture

Input to the generator will be in two sections, a machine description and a language description. The code for most phases of the generated compilation system will be invariant, with language and machine dependences described in data supplied by the generator. Lexical analysis may require a description of the language's lexical structure. Syntax analysis will require a parse table which includes descriptions of connection point calls. Semantics routines may require various information about the language, compilation machine and target machine.

Flow analysis will require information about compilation machine data types for use in constant folding. It will also require a description of possible transformations. Code generation will need descriptions of the target instructions to implement each IL construct and a description of the data types and access modes of the target machines. Memory allocation will require a description of the physical data types and access modes of the target machine.

To produce this information from its input, the generator will need to do several kinds of processing. It must use standard parser generation techniques to produce the needed parse tables. A simpler extraction may be appropriate to give lexical analysis the information it needs. Cattell has described techniques for extracting the tables code generation needs. Simpler analysis and extraction are needed to get the information flow analysis and memory allocation need from the machine description.

This leads to identifiable functions for the generator which can be isolated in modules. A description of the lexical structure may need to be extracted from the language description. Parse tables must also be generated for syntax analysis from the language description. Descriptions of the target and compilation machine types may be needed by the two code generation modules, ORD and CG will need possibly distinct semantics and flow analyses descriptions of the target realizations of TCOL constructs along with their costs.

A description of the storage bases, data types, and access modes must be extracted from the machine description for memory allocation.

4.2 Compiler Architecture

For reliability and simplicity, it is desirable that as much as possible of the retarget compiler be invariant, written once for all language-machine pairs. A great deal of research has gone into identifying the language- and machine-dependent aspects of syntactic analysis, flow analysis, code generation and resource allocation. The retargetable compiler system will exploit this work by isolating the machine and language dependencies of the above processing in data tables, rather than attempting to synthesize appropriate code.

The structure of the retargeted compiler will be somewhat similar to that of the BLISS11 compiler described in [WJW75]. It is pictured in Figure 4.1.

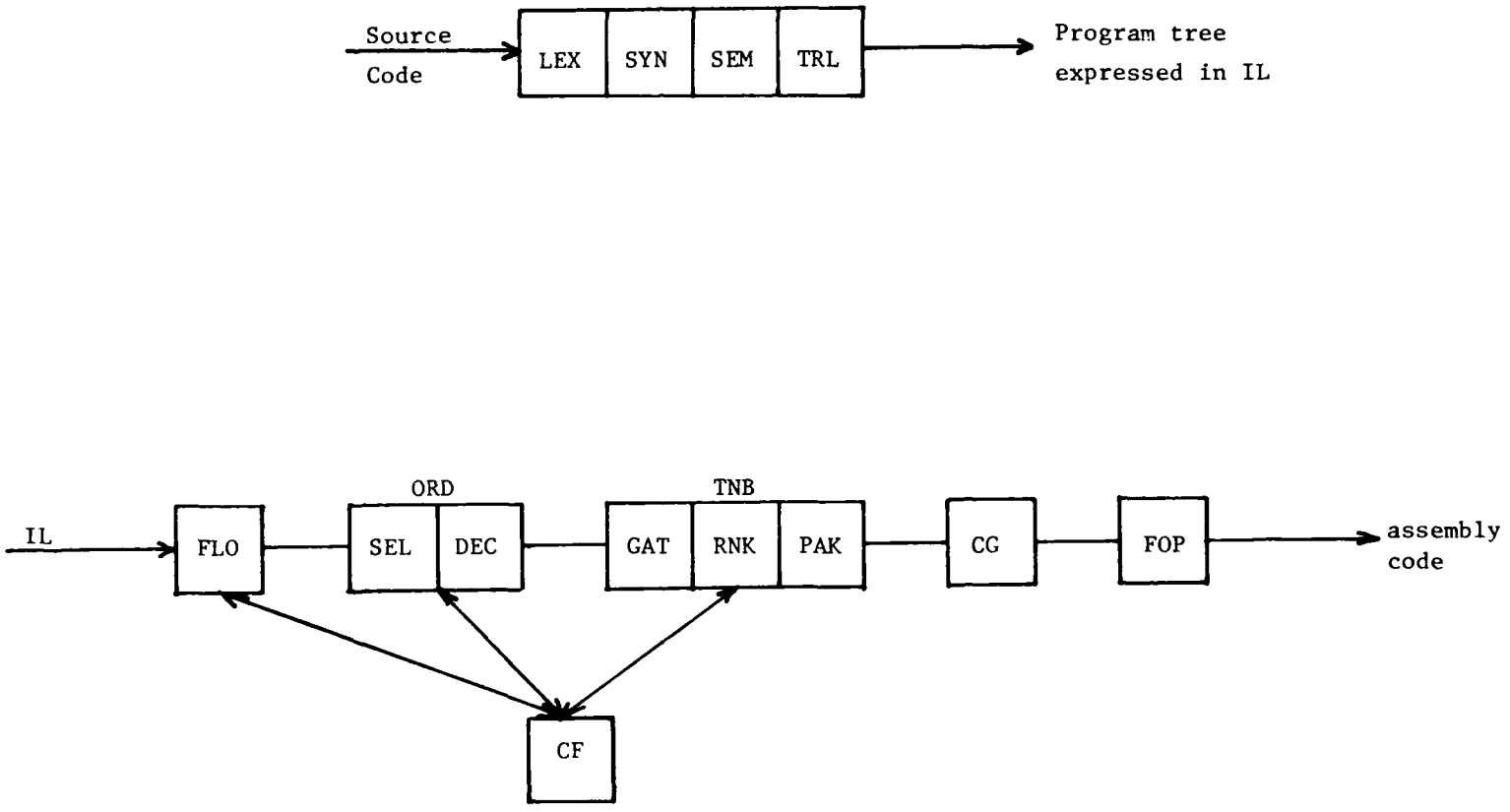


Figure 4.1 Retargetable Compiler Architecture

Lexical analysis, syntactic analysis, semantic analysis and production of the IL form of the program will be handled in a single conceptual pass. These functions are handled by the phases LEX, SYN, SEM and TRL respectively. Flow analysis, evaluation order determination, storage allocation, code generation and "peephole" optimization will each be handled by a single phase in a pass by itself, named FLO, ORD, TNB, CG and FOP, respectively.

To make LEX invariant over all languages would seem overly ambitious, because of efficiency, the difficulty of formally describing a lexical syntax and the dependence of lexical interpretation on context. Fortran and PL/I can be particularly tricky in their lexical structure. In addition, LEX will need to build the tokens which are used by the SEM routines, which are supplied by the implementor, so for LEX to be invariant, it must build tokens that supply all the information that any SEM routines might need. This seems hopeless and unproductive. However, for a class of languages with regular lexical structure these arguments no longer apply and the advantages of using proven routines and a formal description system become more compelling. A middle ground is to have invariant scanners to collect strings of digits, and letters and skip comments and blanks. The implementor can then write a screener that uses these primitives to ease his coding task. Also, one can supply a variety of packages appropriate to specific languages or classes of languages. The choice of lexical analysis strategies is thus dependent on the class of languages one wishes to address and other design goals. Only in one way will LEX differ from traditional lexical analyzers: LEX will not resolve constants which occur in the program. Instead, they will be stored in a string table just as they occur in the program and will be converted as required (in FLO and CF for the compiler computer, in CG for the target).

SYN will be based on syntax-directed parsing techniques. An LALR(1) or LL(1) parser generator embedded in the generator can generate the needed parse tables. SYN can exploit automatic error recovery techniques though it is probably best to avoid techniques that modify the syntactic stack, since this will avoid inconsistent calls on the semantic routines simplifying the implementor's task in writing these routines. Since SYN will only

be written once, it is preferred that it be implemented in a way that eases the writing of more variable parts of the compiler, even if SYN's complexity is thereby increased.

SEM will be a collection of routines supplied by the implementor to do the semantic analysis of the source program. The routines are envoked by SYN as directed by the parse tables, and will determine possessions, check compatibilities, perform operator identification and attribute coordination, and call upon TRL routines to build the IL program tree. SEM will encode in the tree source and target data types and accessing information for the operands in the tree. To support optimization, SEM will also need to encode the logical orderings required by the source language, as specified in the discussion of FLO. Attribute grammars have been proposed as a way of formally describing the semantics of a language and directing the semantic analysis of a program, but this technology does not yet seem well enough understood or efficient enough for use in a practical compiler.

The TRL routines will be invariant and will include generalized symbol table and IL construction routines. The symbol table routines will need to support the processing needs for languages with controlled scope (e.g., Ada, Euclid), explicit aliasing (Fortran), user defined modes, and other features of existing languages. The IL construction will need to support the specification of required orderings, source data types and target data types mentioned above. The implementor's coding burden will be greatly eased by these routines and they will protect the integrity of the symbol table and IL.

For some languages, such as Pascal and Fortran , LEX, SYN, SEM, and TRS could operate in parallel as part of the same pass, with analysis of the program and production of IL proceeding together. For other languages, such as Ada, multiple analysis passes may be required and production of IL must be postponed to the final analysis pass. These passes will be implemented by SEM, leaving SYN and TRL unchanged.

FLO will perform global analysis, doing live/dead analysis on variables, constant propogation and folding, identifying dead code, dead variables, common subexpressions, and finding opportunities for code motion and strength reduction. It will make use of ordering information in the IL. FLO will use the same IL format as input and output, with its functions only being IL transformations, leaving FLO invariant from compiler to compiler.

ORD will make decisions about potential optimizations identified by FLO, determine evaluation order and identify temporary names (temporary results), specifying their lifetimes, preferred target data type, storage base and access mode.

Cattell has described a method discussed below and in [Cat78]called MMM, and a data structure, the LOP, useful for approaching this problem. ORD will output a modified IL tree and a description of required temporary names. TNB will assign target memories to the various temporary names described by ORD. Where possible, TNB will assign multiple temporary names to a single memory.

CG will use the memory assignments calculated by TNB to implement the IL program produced by ORD. Again, the MMM and LOP described by Cattell will be useful. In addition, a temporary storage manager will be required by CG where assignments by TNB make the evaluation order specified by ORD impossible. The temporary manager would save the values in currently allocated but required memories and then restore the old values later, much in the manner of conventional register allocators.

FOP will perform "peephole" optimizations using as its target machine code input and output, correcting inefficiencies in CG's output due to code from widely separated parts of the IL tree becoming adjacent in the produced machine code. These optimizations include elimination of redundant machine operations such as loops, loads and stores, and branches to branches. These transformations are really hard to systemize and will be supplied by the implementor.

4.3 Compiler Module Algorithms

In the remainder of this section we will concentrate on the algorithms and peculiarities of the "backend" routines, namely CF, FLO, ORD, TNB, CG, and FOP. These routines are of the primary interest to retargetable compiler study, since, as noted in the SOW and the introduction front-end theory and practice are well established.

Module Generator:

Input	Processor	Output
syntax grammar of language	LALR(1) compiler-compiler (e.g., YACC)	LALR(1) syntax tables
	Compiler Module:	
source program	- LALR(1) parser with tables	- TCOL program tree
	- user supplied lexical analyzer	- symbol table (including string and constant tables)
	- user supplied semantics routines	
	- TRL tree and table manipulation routines	

Figure 4.2 LEX/SYN/SEM/TRL HIPO Chart

Module Generator
(no preprocessing for FLO)

Compiler Module:

Input	Processor	Output
- TCOL program tree - Symbol table	- user supplied "safety" decision routines - FLO processor - CF subroutine	- TCOL program tree with constants folded, unused code eliminated, common subexpressions threaded, and code motions identified

Figure 4.3 FLO HIPO Chart

4.3.1 FLO - Flow analysis

The FLO module performs data flow analysis on the program tree, performs constant propagation and detects other possible FLO and classical optimizations. The difference between FLO and classical optimizers is that FLO detects possible global optimizations, but the desirability of performing a given transformation is determined later, by ORD. This is the technique used by Wulf et al. in the BLISS11 compiler.

A global optimizer needs to determine the lifetime of each variable or computation (data flow analysis), and to determine feasible optimizations, which are motions or eliminations of computations in the program. Most of the work performed by the optimizer is language independent. Language dependencies arise in two ways: limitations on the lifetime of a variable and limitations on when code motion is legal given the semantics of the source language. To move or eliminate computations, the optimizer must determine their lifetimes, and whether or not relationships between objects

can be changed in a given situation. As an example of the effect of language semantics on lifetimes, a function call in Fortran may alter any variable in COMMON. As an example of the effect of semantics on possible optimization is that in Fortran association and commutation may always be applied to multiplication; in Algol this is not true. These effects have to be taken into account in the optimizer for a given language, and in FLO user-provided procedures will do this. There are a number of formulations of global analysis in the literature, some quite recent. The one that will be used in this discussion is the one developed by Geschke [Ges72] in his thesis and used in the BLISS11 compiler. Its advantage is that it makes the isolation of language dependent information simple.

According to Geschke, the language dependencies of FLO can be described by three relations called "initial order", "necessary constituent", and "essential predecessor". These encode the information described above, and are used to derive sets of data flow information which are used to determine feasible code motions and code eliminations. Procedures to determine these relationships will be supplied by the compiler writer.

FLO will input a program tree, and proceed down the tree, detecting common subexpressions and determining the data flow sets mentioned above, which are then used to determine possible optimizations. Outputs of the phase are the program tree with constant folding and dead code elimination performed and lists of common subexpressions and feasible optimizations. These optimizations are those mentioned in the section on optimization techniques (3.2.3).

4.3.2 CF - Constant Folding

The constant folding routine is unique in that it is repeatedly called by FLO, ORD, and CG in order to perform constant computations which appear in the program tree. CF is recalled in each of the routines to perform those computations uncovered by adjustments made to the tree by other optimizations.

Module Generator:

Input	Processor	Output
- data description portion of MOP description		- procedures for bit string to/ from char string conversion routines (for compiler machine)

Compiler Module:

- program tree - symbol table	- CF subroutine - Conversion routines	- program tree, with constant expressions folded - symbol table suitably updated

Figure 4.4 CF HIPO Chart

Module Generator:

Input	Processor	Output
- MOP description	- as described in [Cat78]	- condensed table (termed LOP)

Compiler Module:

- program tree from FLO - LOP	- ORD routine - CF subroutine	- optimized program tree including preferencing attributes

Figure 4.5 ORD HIPO Chart

4.3.3 ORD - Tree Ordering

The ORD module's purpose is primarily to perform a preliminary code generation, in order to collect the data required to intelligently select among the code movements found possible by the FLO module. ORD also produces and distributes a number of code-generation derived attributes around the TCOL tree, for use by the following TNB and CG phases.

The preliminary code generation phase, called SEL, uses an algorithm named by Cattell the "Maximal Munching Method (MMM)" algorithm [Cat78]. In essence, this algorithm attempts to generate assembly code for the largest part of the code tree (from the top downward) at a time, and then recurse upon the remaining unmatched branches. Inasmuch as register allocation has not yet been done the match must make assumptions concerning allocation of the data leaves and temporary locations generated which may prove to be false; however, the need is for approximate time/space values for comparison purposes, so the assumptions made will probably cancel out.

For each code movement possibility indicated by FLO, a time/space comparison will be made, and if the code movement is an improvement, the movement will be marked. The total time and space accumulations (as well as other data, such as probable number of executions, etc.) will be passed to a user supplied tradeoff routine which returns some measure of quality, thus giving the user control over the goals of the optimization. In a final pass, the tree will be restructured by performing the indicated code movements.

The second phase of ORD, called DEC, performs some kinds of machine-dependent attribute determinations and tree decorating. These attributes include self-complementing operations (i.e., x = -(-x)), use of special register subsets (i.e., the M instruction on the S/370 places its result in an even-odd register pair) and taking advantage of the machines effective address capabilities to perform arithmetic.

The decoration of the program tree with the "negated" attribute will allow elimination of redundant negation operations within the tree (note that "negation" is used herein generically to indicate any self complementing operation). At any given node in the tree, each of the branches to that node will report back the value of the negation attribute of that branch. The DEC pass, upon arriving at that node in an end-order walk of the tree, will use these values to compute the "best" value of the attributes to be passed up the tree to higher nodes. This computation is generally based on the concept of performing the minimum number of negations in order to satisfy the node's operation.

The pass also has the ability to eliminate negation nodes, to change addition to subtraction (and vice versa), and to reorder branch phasing in order to minimize operations. An example of negative propagation might input the tree:

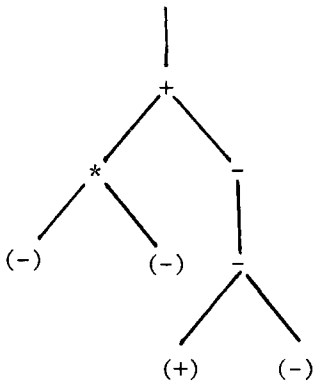


Figure 4.6 ORD Example Input

where the parenthesized signs represent the values of the negation attributes at the various nodes. After negation attribute propagation, the tree would resemble:

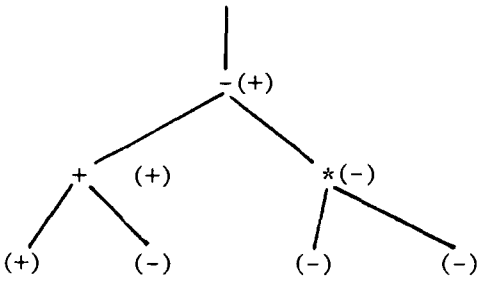
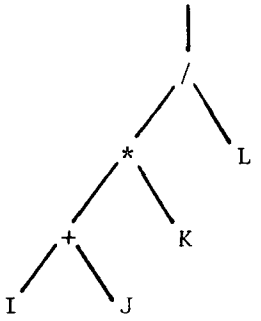


Figure 4.7 ORD Example Output

A similar (and simultaneous) operation would decorate the tree with attributes which indicate the desirability of having operands to particular operations computed in certain registers. These particular operations may include, for example, integer multiplication and division on some machines, which require their inputs in a subset of the available registers. For example, the following tree on a machine like the IBM S/370 might produce the code indicated.




```
LD RO,I
AD RO,J        ; RO=I+J
MV R1,RO       ; move to odd reg
MP RO,K
DV RO,L
```

Figure 4.8 Register Preference Example

However, the MV instruction could have been eliminated if the ADD node had known the result was required in an odd register.

Finally, the DEC routine will search for those nodes in the tree which could be subsumed in the effective address computation of the machine's hardware. For example, the tree:

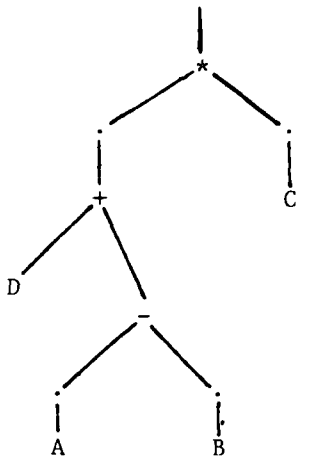


Figure 4.9 Addressing Mode Example

could be performed in two instructions on many machines if it is recognized that the result of the substract operation should be bound to an index register, and thus the add operation may be performed as part of the indexing operation in the multiply operation. DEC will thus indicate in the attributes of the referencing operator above the add operation the strong preference for its input to be in an index register.

The negation attribute discussed above will be of no further use once ORD is complete; however, the register preferencing attributes are needed by the register allocation phase (TNB) and will remain until then.

Upon completion of ORD then, the program tree will be in its final "shape", reflecting the final layout of the program. Further phases of the compiler will not change that basic shape, but rather change node values and attributes.

Module Generator:
(see Figure 4.5)

Compiler Module:

Input	Processor	Output
- program tree from ORD - LOP	- TNB routine	- program tree with locations and registers assigned to temps, variables, and constants

Figure 4.10 - TNB HIPO Chart

4.3.6 TNB - Temporary Name Binding

The TNB module has the function of binding all user variables, local, global and temporary, to real computer resources in such a way as to insure the integrity of the program semantics, and to provide the "best" use of those resources. In reality, TNB must balance the requirements of fast access and the usually limited number of high speed data locations.

Historically, register allocation has been one of the really tough computational problems in compiler theory. There are no algorithms currently known which will guarantee optimal allocation in all usual compilation circumstances, and many of the better algorithms can become extremely time consuming when some worst cases are attempted. Obviously, what is needed for now is an algorithm which makes a minimal sacrifice to optimality while becoming well behaved in a computational sense.

The register allocation scheme of Johnsson [Joh75] seems to fill these requirements, and in addition, was formulated with requirements for machine independence.

Johnsson's algorithm is divided into three phases, which we call GAT, RNK, and PAK.

GAT is an information gathering phase; it passes over the tree, assigning each address-valued node a temporary name (TN) representing the location where its value can be temporarily stored. In addition, references to TNs and other variables are noted, including whether the value is changed or simply used. When the pass is complete, a list of "lifetime pairs" is computed for each TN (i.e., a pair of points in the program between which the TN is "alive", or in active use). Finally, preferences for particular storage bases (such as those declared by the access mode determination part of ORD) are noted.

The RNK phase performs the primary analysis of the TNB module by building an interference and preference graph for the program being compiled. The interference graph links those TNs which have overlapping lifetimes. Thus, two nodes which are connected (over any path) in the interference graph may not be assigned to the same physical location. The preference graph links those TNs which should be assigned the same location in order to avoid extra load and store instructions. These links may be weighted to express, for instance, an increased preference for TNs within loops.

Finally, the PAK phase distributes the known machine resources to the TNs by performing the "packing algorithm". There is not yet an optimal packing algorithm known. Many different algorithms and heuristics are employed at this time, and further research is being done on the problem. A good algorithm must meet four basic criteria (from Leverett e. al. [LCH79]):

"- No two TNs which are connected by an interference arc may be packed in (allocated to) the same storage location.

- The cost measure determined by summing the relative costs of all TNs, as derived from the usage information discussed in previous section, and from the knowledge about which storage class each TN has been packed in, should be kept low (perhaps minimized).

- The profit measure determined by summing the values of all preferences arcs that connect two TNs packed to identical locations should be kept high (perhaps maximized).

- For some storage classes, there may be a cost associated with using any member of the storage class, which is fixed regardless of how the member is used. For instance, a run-time convention for the preservation of register contents across routine calls may require that if a register is used by a routine, it must be saved at the beginning of the routine and restored at the end. Thus there is a cost measure determined by the number of locations (of certain classes) which are used in a given packing; this should be kept low.

In a final pass, PAK will distribute its packing decisions throughout the tree, replacing variables and TNs with register numbers or other storage base locations. The program tree has now been allocated in memory and final code generation may take place.

Module Generator:
(see Figure 4.5)

Compiler Module:

Input	Processor	Output
- program tree from CG - LOP	- CG routine	- linked list of instructions

Figure 4.11 CG HIPO Chart

4.3.5 CG - Code Generation

The code generation phase performs the final pass on the program tree, converting it to actual assembly language code. It attempts to build locally optimal sequences taking full advantage of the instruction set and effective address calculations.

The primary algorithm used to perform this task is Cattell's MMM algorithm, mentioned earlier in the discussion of the ORD module. This algorithm attempts to pattern match the root of the program tree against an ordered series of trees built by the retargetable compiler generator. This list of trees is ordered in such a way that the least expensive special case instruction sequences are searched first in order to satisfy the requirements of a given tree pattern. For instance, if the tree root was an add operator, the first code segment to try could be an increment instruction, then an add immediate, then an add register. Since address references were recognized in the ORD phase and appropriate TNB assignments requested, most such calculations should match effective address computations represented by the MOP's access modes. Finally, all unmatched subtrees are in turn examined by MMM as it recurses on each of them.

As an alternative code sequences are found for each node in the tree, within the framework of the tree and register assignments made previously total time/space statistics are compiled and compared, insuring that the optimal local code is selected. This code, when output in order by a final end-order tree walk, may still be subject to certain code level optimizations, which will be the subject of FOP's gentle ministrations.

4.3.6 FOP - Final Optimization

After CG has determined which instructions are to be generated, FOP will perform final optimizations and emit machine code. Many of these final optimizations are so machine dependent that no formalism for FOP exists, and parts of it will have to be coded by hand. Some of these optimizations

Module Generator:
(none for FOP)

Compiler Module:

Input	Processor	Output
- list from CG	- FOP processor - user supplied routines	- optimized list of instructions in assembly code

Figure 4.12 FOP HIPO Chart

however, are nearly machine independent, and others fall into general classes, so that a framework for FOP can be provided, which must then be tailored by hand for a given machine. In general, the structure of FOP is similar to the final pass of the BLISS11 compiler.

The first sub-phase of FOP performs optimizations which involve multiple (and possibly widely separated) instructions. These include cross-jumping store/load pairs, etc. Since, in some cases, these optimizations can make other improvements possible, this phase will be repeated until no optimizations remain to be performed. The second sub-phase will examine each instruction once, trying to replace it with a cheaper but equivalent instructions.

The third sub-phase of FOP generates the final compiler output. For machines, such as the PDP-11, with "span-dependent instructions" such as long vs. short form jumps, the proper alternate form will be selected here. T. G. Szymanski's algorithm, which is efficient and optimal [Szy78] will be used to minimize the length of these instructions.

4.4 The MOP Computer Description

The MOP is a functional machine description in terms of input-output assertions. That is, it maps inputs in the form of tree templates into outputs which are all machine instructions (in assembly or machine code). The description itself is in LISP list notation in which the tree templates may be easily encoded. For instance the notation:

```
(; (← $1 (- $1 1)) (← %N (LSS (- $1 1) 0)) (← %Z (EQL (- $1 1) 0)))
```

is a linearization of the tree:

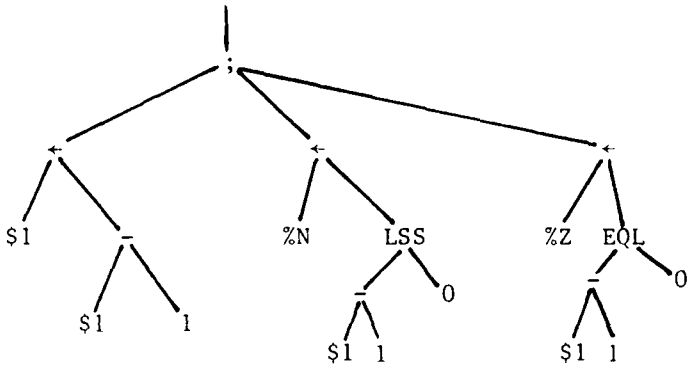


Figure 4.13 LISP notation

The semicolon operator indicates a series of alternatives; in this case, either of the three patterns, if matched, will output the same code. The leftmost sub tree indicates the direct action of a decrement instruction (in which the parameter "$1" must match a register), while the other two indicate the settings of condition codes %N and %Z by the decrement action. Note that each pair of subtrees represents the side effects of applications of the third.

The MOP is a set of six tables which describe different aspects of the machine:

Storage Bases (SB) - a list of memory components in the machine, their sizes and uses;

Instruction fields (I-flds) - a list of fields used within instructions, their sizes, displacements, and types;

Access Modes (AM) - a list of ways in which the various storage bases may be accessed (e.g., direct, indexed, etc) in symbolic form;

Operand Classes (OC) - a list of sets of access modes, any one of which may be applicable to a given instruction field, and the cost and format data for the use of each;

Instruction Formats (I-Fmt) - a list of possible formats which an instruction may be written in;

Machine Operations (M-op) - the instruction I/O assertions, including cost and format data for the use of each.

There is also a quasi-machine independent table used with the MOP description, called the Axiom List. This is a table of transformation functions which allow a machine with a less-than-comprehensive instruction set to use alternatives. For example, the axiom list includes DeMorgan's laws, definitions of AND and OR operations in terms of each other, etc.

A set of examples taken from a description of the PDP8 computer in Cattell [Cat78] will be presented. The complete description and the Axiom List is contained in Appendix C.

- An entry in the I-Flds list:

  (OP 0 3 0 O)

  describes the field called "OP", which occurs at bit 0 word 0 of the instruction, 3 bits long, of type "opcode".

- An entry in the SB list:

  (Mp 256 12 M)

  indicates the Mp (primary memory) is 256 12 bit words of type "memory".

  Another example is the program counter:

  (PC 1 8 P)

- An Access Mode:

  %Mp: (· · Mp $1:#8 0 12)

  indicates that the AM "%Mp" is a direct fetch from Mp of 12 bits from a constant 8 bit location supplied by the tree.

- The Operand Class "Y" is defined by:

  Y: (%8 :: (EMIT [5 0 0] $1 0)
  %Mp :: (EMIT [5 1 0] $1 1))

  That is, wherever the OC "Y" occurs in a M-op, it can be matched either by a tree leaf representing an eight bit constant (AM is %8) or by a direct memory access (AM is %Mp). Associated with each is a format number (5) and space/time cost "0 0" and "1 0" respectively - note that use of direct access rather than immediate costs one extra word). The final two items for each are formatting templates.

- An example M-op:

  (IF (LSS %Acc 0) (← %PC (+ %PC 1))) ::
  (EMIT [SKPL 3 1 1] 7 1 4)

  This M-op could be called by matching it to the program subtree:

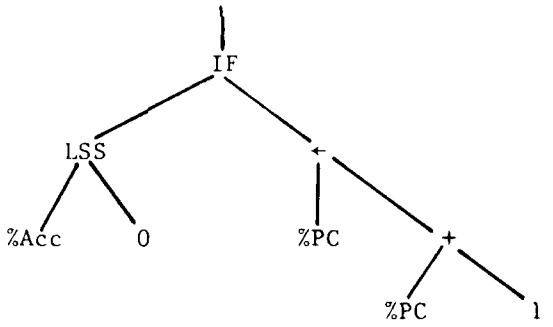


Figure 4.14 M-op Matching

If this were to happen, a SKPL instruction would be emitted, at a space cost of 1 and time cost of 1, using I-Fmt number 3. The "7 1 4" are the contents of 3 of the fields in the instruction, fixed by the selection of this particular operation. Note that the SKPL is actually superfluous; the op code is emitted directly into the I-Fld. If input is to go into an assembler, the alphabetic op may be required.

One final section of machine description data must be added to the MOP. We need descriptions of the data objects used by the machine 'Cat76'. For example, suppose M-ops are supplied for both floating point and integer arithmetic operations. If this is so, then we also need a method to convert floating point and integer language objects into the appropriate bit strings and vice versa.

There are two approaches. We could parametize all the conceivable data types and use a general purpose conversion routine, or we could require user provided conversion routines.

We will take a middle path. For some common data types (fixed point, long and short floating point, strings) we will provide parameterization; for others (or for instances of the former which don't fit the parameters) the implementor will be required to provide conversions. Note that operations upon the data types are provided through the M-ops.

4.3 The TCOL Intermediate Language

TCOL (Cat's, SYM 4) is a family of related intermediate languages, in which each member language is tailored for a particular source language. Thus, $TCOL_{Ada}$ is tailored for the Ada language.

A TCOL representation is an abstract syntax tree in which internal nodes are operators and the external nodes are symbols or constants. TCOL may be expressed in tree notation, as we have been doing in this report, or it may be expressed in a linear form, by LISP notation.

In practice TCOL requires the existance of a symbol table to define the symbols in the tree and their attributes. To these are later added the temporary names defined at each tree node.

In the example in section 5, we will use the following operator :

Operator	Meaning	#opds	Value	Example
;	sequence	*	none	(; S_1 S_2 .. S_n) perform statements S_1 through S_n in sequence
.	dereference	1	value	(. A) contents of location A .
←	assignment	2	none	(A (. B)) is A = B (Fortran)
+,-,*	binary arithmetic	2	value	(+ (. A) (. B)) is A + B
-	unary arithmetic	1	value	(- (. A)) is -A
CALL	procedure	1 + *	none	(CALL X A B) is CALL X (A, B)
FCALL	function call	1 + *	value	(same as CALL)
UPLOOP	increment	4	none	(UPLOOP I J K S) is FOR I : = J TO K S; (Pascal)
DOWNLOOP	decrement	4	none	(DOWNLOOP I J K S) is FOR I : = J DOWNTO K S;
CASE	case construct	2 + *	none	(CASE I S_1 S_2 ... S_n) is CASE I OF (SEL) . . .
SEL	case alternative	2	none	(SEL V S) is (CASE I S_1 S_2 .. S_n) V : S . . .

* - 0 or more

Figure 4.15 Sample $TCOL_{Ada}$ Operators

5. CONCLUSIONS

The theory and practice of compiler development has come a long way since the first FORTRAN compilers. The theory of lexical and syntatic analysis have been advanced, and the production of those portions of the compiler have been automated. We believe that we may now also automate the processes of machine independent optimization, register allocation, and code generation. Another area, machine dependent optimization, may be at least partially automated, but many of these optimizations are still too machine-eccentricity dependent to allow a program to be able to predict what to implement and how to go about it. (It is also hard for human compiler implementors, and very subject to experience on a particular machine.)

Semantics analysis is the other compiler process which does not yet appear amenable to strictly automatic analysis and generation. In this case, however, the theory is advancing and the ability to completely specify language semantics, and therefore the ability to generate the appropriate output from such a specification, seems to be a near future possibility.

We conclude that it is possible at this time to build a comprehensive compiler generator which could, given a language specification and a machine description, generate a compiler for the language on the machine. At this time, it would be necessary to hand code the semantics analysis and at least part of the final optimization pass, as well as some parts of a run-time library. We foresee no intractable problems generating an Ada or J73/I compiler using this generator.

There are many areas which still require study in this field. Besides continuing work to automate the semantics analysis and final optimization passes, the following are topics on which further research probably will yield good results:

- The packing algorithm used in TNB is not optimal, but rather heuristic. A lot of pure topological research is going on to solve the "graph coloring" problem, of which this is an example.

- Run-time monitoring [Knu73], in which the run-time system determines what parts of the program execute the most frequently, should be examined, both as a feedback compiler-writer's aid, and to drive.

- Run-time optimizations, wherein the run-time statistics drive optimization in an effort to increase the in-core efficiency of a program which will be used repeatedly. One really exciting variation on this theme involves the optimization of micro-code supporting the application to be optimized, following run-time analysis of the application. The microcode could be tailored for a particular application in this manner.

- The inclusion of specific machine features, such as paging, cache memory, hardware stacks and queues, etc., whose impact on optimization strategies is not yet well understood.

- Procedure integration and identification, which attempts to optimize across a specific space/time tradeoff by isolating redundant code sections into procedures, and the converse of expanding procedures in-line.

- Machine independent post optimization in which a MOP or similar description is used to drive a general optimizer. [Fra79] describes a general optimization which does most of the work of FLO, this and similar techniques should be examined.

6. A DETAILED COMPILATION EXAMPLE

We will, in this section, attempt to pursue a nontrivial example through the various stages of processing within a retargeted compiler. The example is written in PASCAL and is not meant to be a functionally meaningful program (see Figure 6.1). After the front-end pass, it has been expanded into an equivalent TCOL tree with accompanying symbolic information (see Figure 6.2).

There are several things to notice in this front-end conversion. The tree is of course built by the TRL routines under the direction of the user supplied SEM phase. Note that the array accesses have been expanded, by making assumptions covering both the language interpretation of arrays and the machine. The symbol table has been consulted, in order to find the value of the lowest array bound of each array.

The semantic routines could have as easily prepared a call to a general array referencing subroutine, if the language (for example) allows dynamic arrays.

The symbol table (Figure 6.3) initially contains variable types and bounds (for arrays) and values for constants.

When FLO is called, several things happen. Constant folding is called, flow analysis identifies constants and dead variables, and the tree is threaded with common subexpressions. These changes are shown in Figures 6.4 and 6.5.

Figure 6.4 is unchanged from 6.2, except for the addition of common subexpression chains. These chains will be used by ORD in eliminating redundant computations. Figure 6.5 shows the possibilities for constant folding, constant propagation, code motion and strength reduction. The "+0" terms will be removed when they are recognized, and the constant selector

```
VAR:  A:  ARRAY  [0..10]  OF INTEGER;
      B:  ARRAY  [0..15]  OF INTEGER;
      C, F, D:  INTEGER;
      Q, R, K, N, LCOUNT, MTYPE, E, WL:  INTEGER;
CONST:  MTYPE = 360;
BEGIN
LCOUNT: = 5; R := 0; C := 0; D := 0; F := 0; E := 0;
READ (Q,N);
FOR K := N DOWNTO 0
    BEGIN
    R := R + A  [2 * K] ;
    C := C + A  [3 * K + 1]  + 2 + N * ABS (Q);
    F := F + B  [3 * K] ;
    FOR I := 0 TO K
        BEGIN
        D := D + I + C;
        E := E + LCOUNT;
        END
    D := D * C;
    END
 CASE MTYPE OF
     360:  WL := 32;
    1130:  WL := 16;
    6000:  WL := 60;
 END
WRITE ('WORDLENGTH IS', WL)
```

Figure 6.1 Example Code Fragment

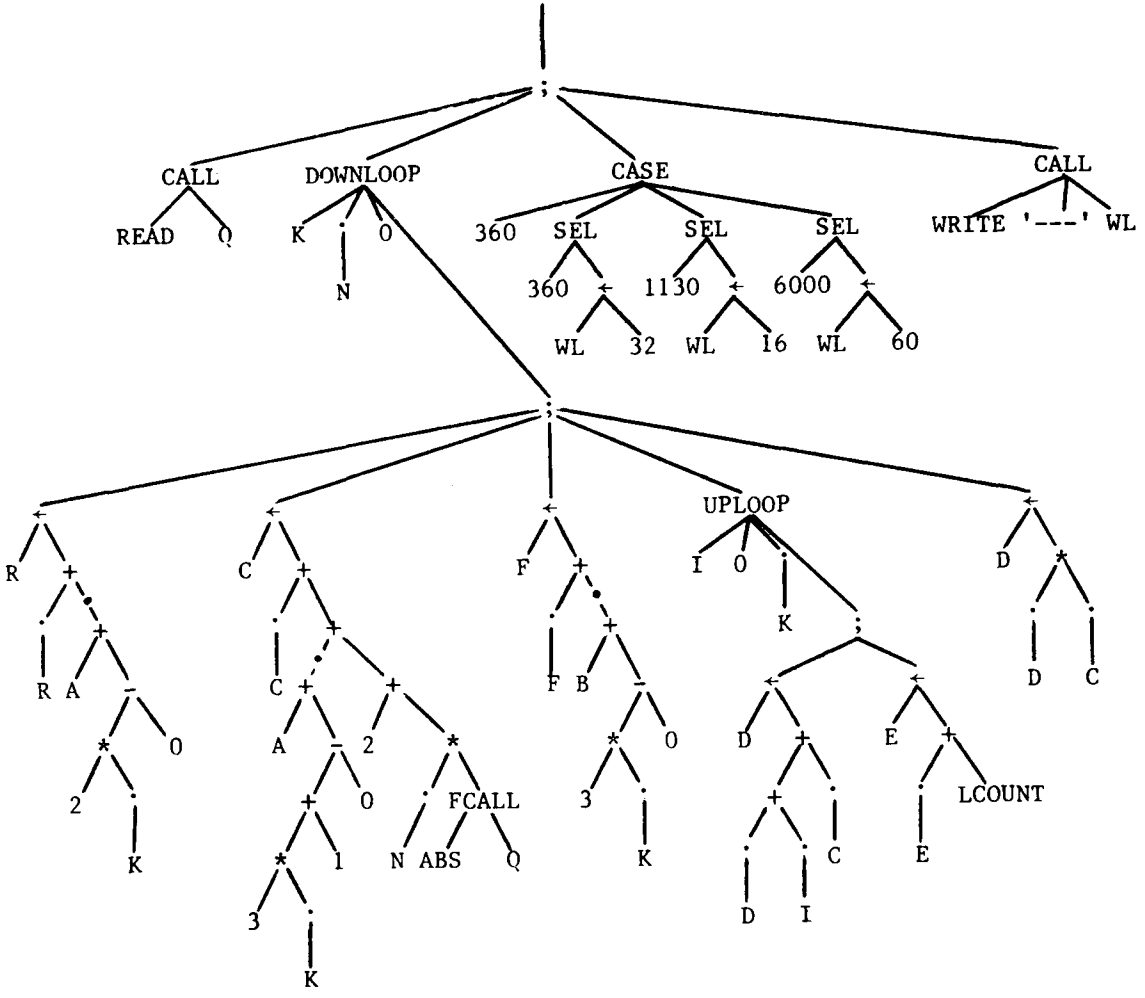


Figure 6.2 - Initial Program Tree

Name	Type	Bounds Low	Bounds High	Value
K	INT			
N	INT			
B	INT ARRAY	0	15	
A	INT ARRAY	0	10	
C	INT			
F	INT			
I	INT			
D	INT			
E	INT			
LCOUNT	INT			
MTYPE	INT CONST			360
WL	INT			
R	INT			

Figure 6.3 Initial Symbol Table

for the "CASE" expression will cause it to be collapsed to (← WL 32). Since this is the only assignment to WL, all occurrences of WL will be replaced by a constant 32 and this assignment will be eliminated. It is recognized that "2 + N * ABS(Q)" is invariant with respect to the loop it appears in, and its possible motion out of the loop is passed to ORD. Finally, "2 * K" and "3 * K" are recognized as candidates for strength reduction.

Figure 6.6 shows the program tree after ORD has determined and performed feasible optimizations. The expressions "2 * K" and "3 * K" in the outer loop have been replaced by subtractions, with T1 and T2 being initialized outside the loop, and the expression A + 1 has been replaced by a constant with the value of A + 1 (note that "A" is a location constant). T3 is set to the value of the loop-invariate expression outside the loop.

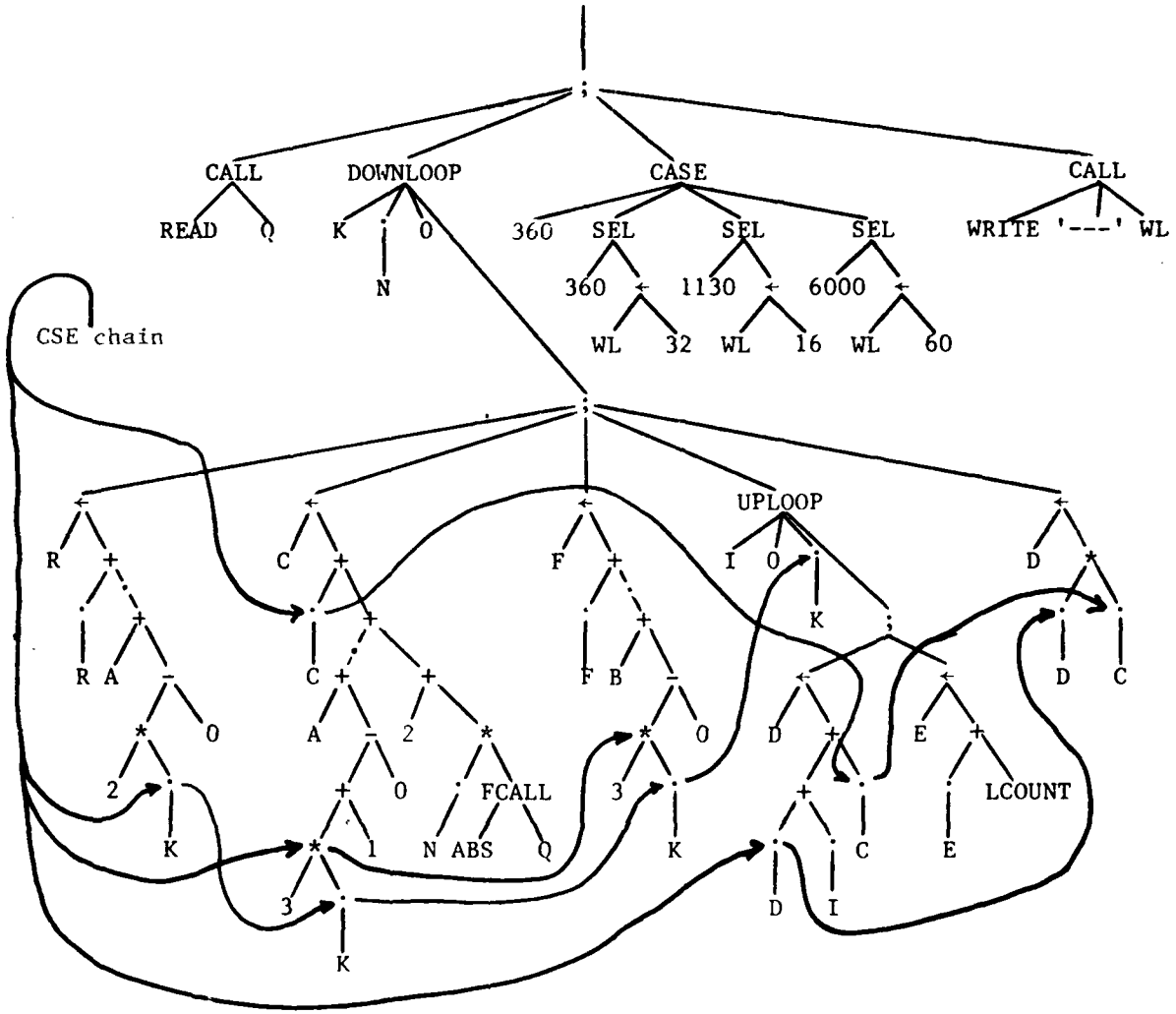


Figure 6.4 - Common Subexpressions Linked

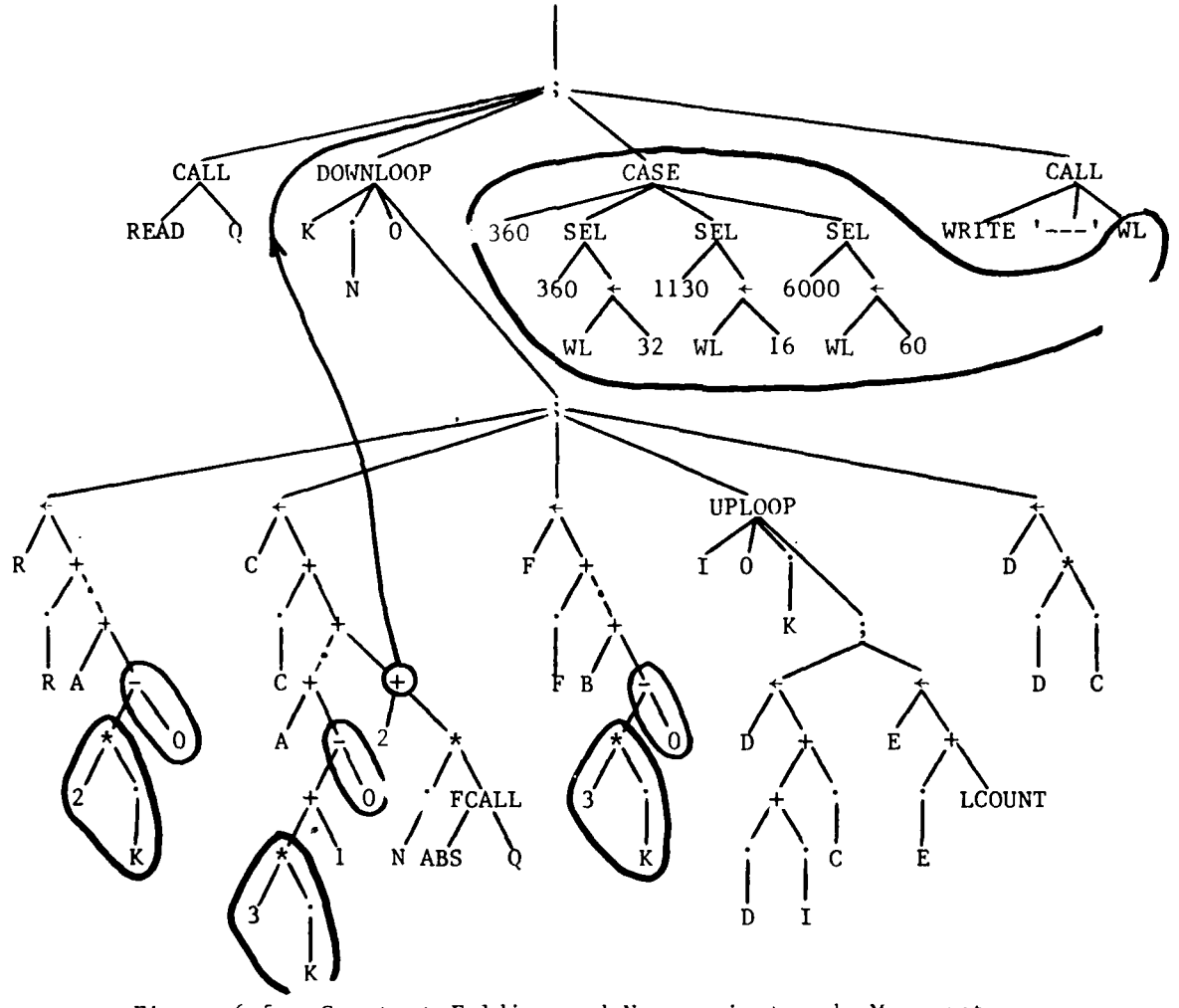


Figure 6.5 - Constant Folding and Non-variant code Movement

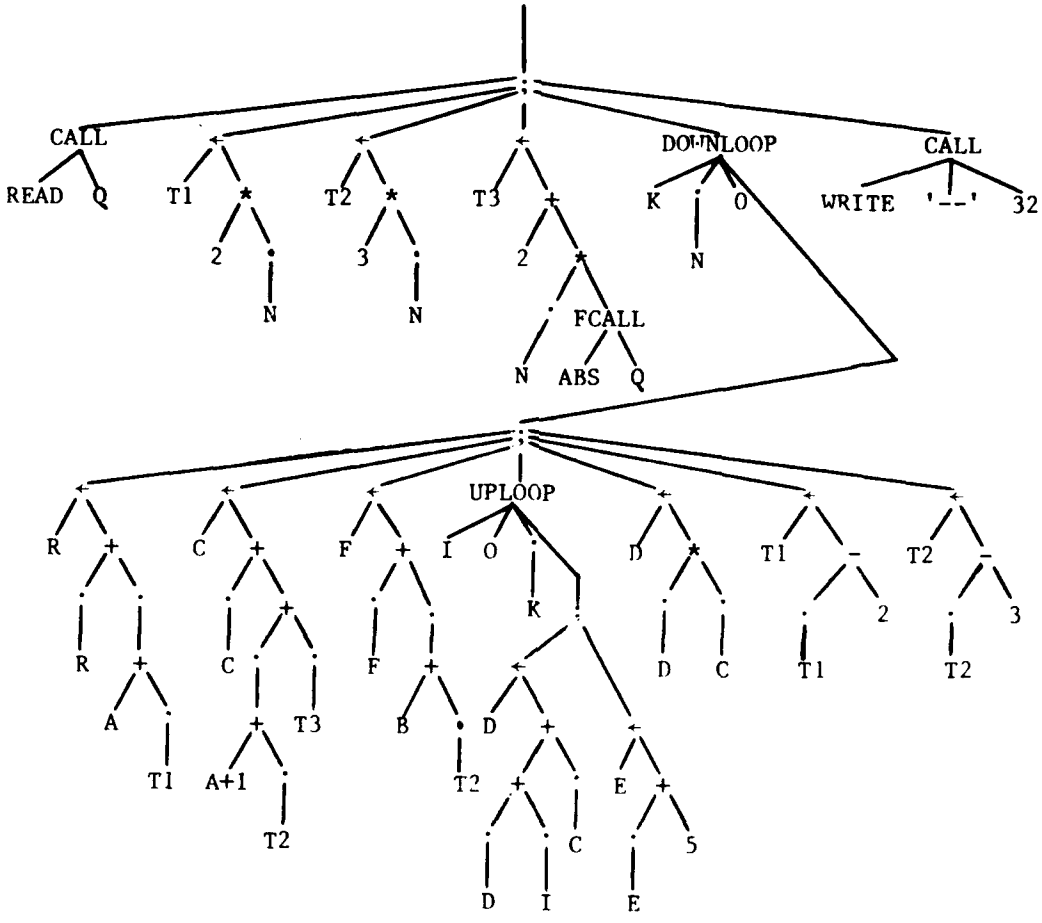
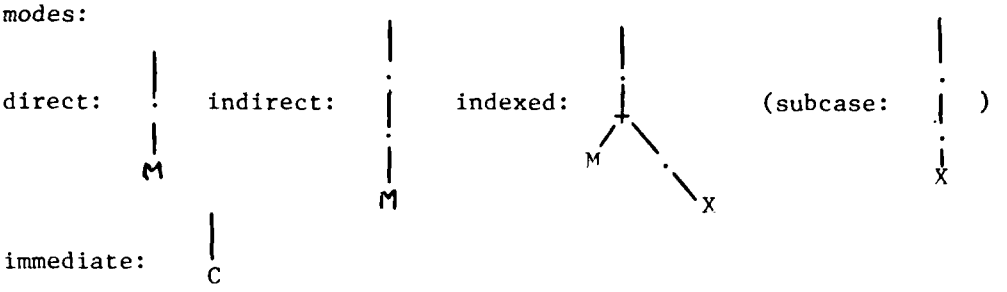


Figure 6.6 - Optimized Tree

ORD must finally try to set some attributes in such a way that optimal use is made of the machine's addressing hardware. If we allow the following access modes:



where C is a constant, M is a memory address constant (possibly relocatable) and X is an index register constant. Access Mode determination will search the tree for such patterns and apply attributes (and possibly perform some rearranging) in order to save instructions. The result is that T1 and T2 are bound to index registers, as also shown in Figure 6.7.

For this example we will define the following properties for the integer instructions:

add: $r \leftarrow r + x$

sub: $r \leftarrow r - x$

mul: $r_p \leftarrow r_o * x$

div: $r_o \leftarrow r_p / x$

where r is any register, x is any register or memory location, r_p is an even-odd register pair, and r_o is the odd register of that pair. In order to effect proper register preferencing, we will use two attributes to each node of the tree:

preference/$Pref_x$: = odd/O, even/E, pair/P,
don't care/D, memory/M

result/Res_x : = O,E,P,M, any register/R,
register 2/2

At any given node, then, the attributes of result will be passed up the tree, while preferences for the operands will be passed down, while the attempt will be made to match the result attribute to the requested preference. We can then formulate the computations on the attributes as a function of the op code:

$*$: $Pref_L \leftarrow Pref_R \leftarrow 0$, $Res_U \leftarrow P$

/ : $Pref_L \leftarrow P$, $Pref_R \leftarrow D$, $Res_U \leftarrow 0$

These two are relatively straightforward, since the requirements are strict. Multiply requires at least one of its operands in an odd register, and returns a pair; the dividend must be in a pair, the divisor may be anywhere, and the result will be in an odd register.

: $Pref_L \leftarrow Pref_U$, $Pref_R \leftarrow D$

$Res_U \leftarrow$ IF Res_L = M THEN $Pref_U$ ELSE

(IF Res_L = P THEN 0 ELSE Res_L)

+ : $Pref_L \leftarrow Pref_R \leftarrow Pref_U$

$Res_U \leftarrow$ IF Res_L = $Pref_U$ THEN Res_L ELSE

(IF Res_R = $Pref_U$ THEN Res_R ELSE

(IF $Res_L \neq$ M THEN Res_L ELSE

(IF $Res_R \neq$ M THEN Res_R ELSE

(IF $Pref_U$ = P THEN E ELSE

(IF $Pref_U \neq$ D THEN $Pref_U$ ELSE chose arbitrarily)))))

The latter two productions for result attributes are more complicated because the instructions are more flexible; they may leave results in many different places, depending on where their operands are located.

Finally, we need to determine productions for some other operators:

$$
\begin{aligned}
* &: \quad \text{Pref}_R \leftarrow D \quad \text{(no result)} \\
. &: \quad \text{Pref}_D \leftarrow D,\ \text{Res}_U \leftarrow R \\
\text{CALL} &: \quad \text{Pref}_R \text{ (for each R branch)} \leftarrow M \\
\text{FCALL} &: \quad \text{Pref}_R \leftarrow M,\ \text{Res}_U \leftarrow 2
\end{aligned}
$$

The last referring to the system conventions that FCALL leaves its value in register 2 in all cases. When the attributes are distributed over the tree (i.e., it is "decorated") the result is Figure 6.7. Note that there are several disagreements; however, all but one result may be presented in a compatible way to the preferences - the one non-compatibility results from the result of the FCALL being in register , whereas an odd register was desired. This incompatibility will be relieved during code generation.

Register allocation will now attempt to bind the various nodes to machine locations. Figure 6.8 shows the tree at this point, decorated with temporary names and with basic blocks delineated. The attribute analysis made earlier determined that Q_8 must be R2 and that Q_2, Q_4, and Q_{37} must be odd numbered registers. Furthermore, Q_{14}, Q_{20} and Q_{26} must be index registers.

The initial pass of register allocation now proceeds through the tree, collecting lifetime data on all the variables and temporaries, and then the later pass allocates on the basis of that data. In our example the allocator might find that it would be best to keep the loop indicies in registers. If the machine had four registers (R0 through R3), and R1, R2, and R3 could be index registers, then a possible allocation is shown in Figure 6.9. All of the nodes remaining unmarked are internal to effective address computations.

Code generation makes the heaviest use of the machine description provided by the user. We will suppose the L-ops shown in Figure 6.10 have been provided. The MMM algorithm now proceeds to try to match the highest nodes on the tree to patterns in the LOP table. The CALL node is

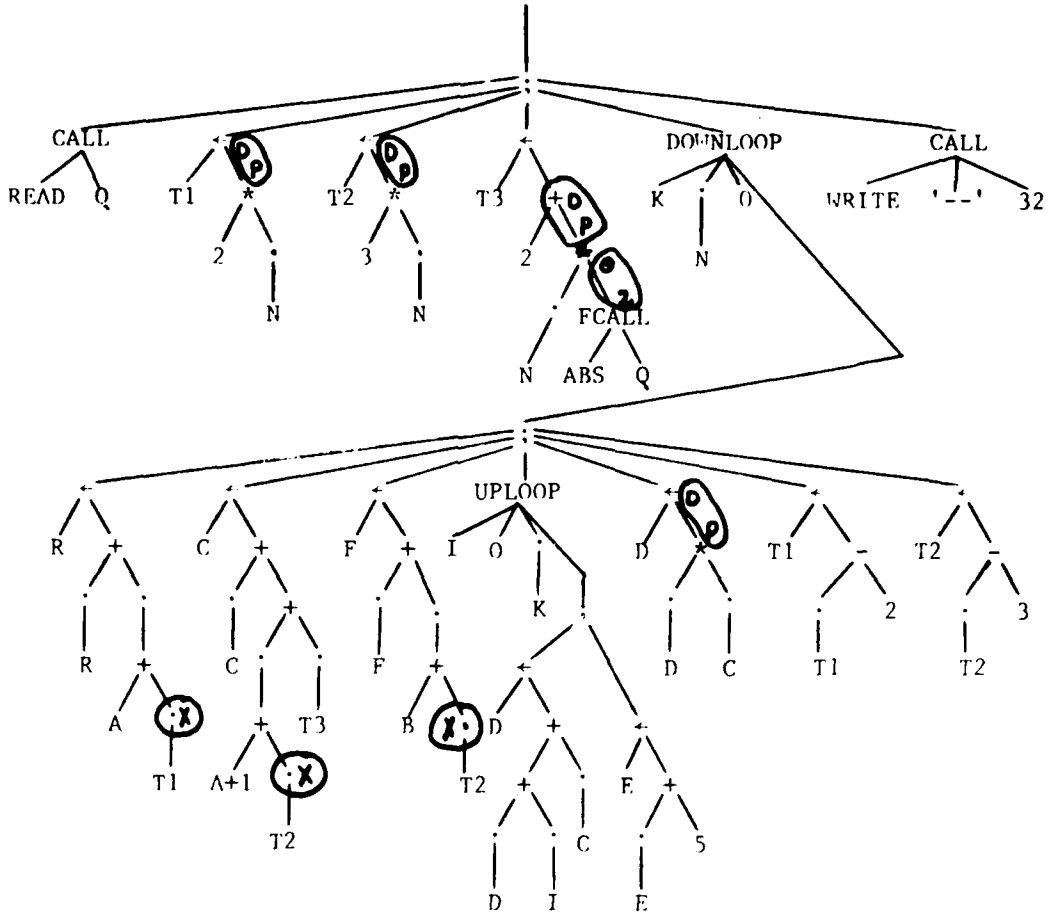


Figure 6.7 - Attributes and Access Mode Determination

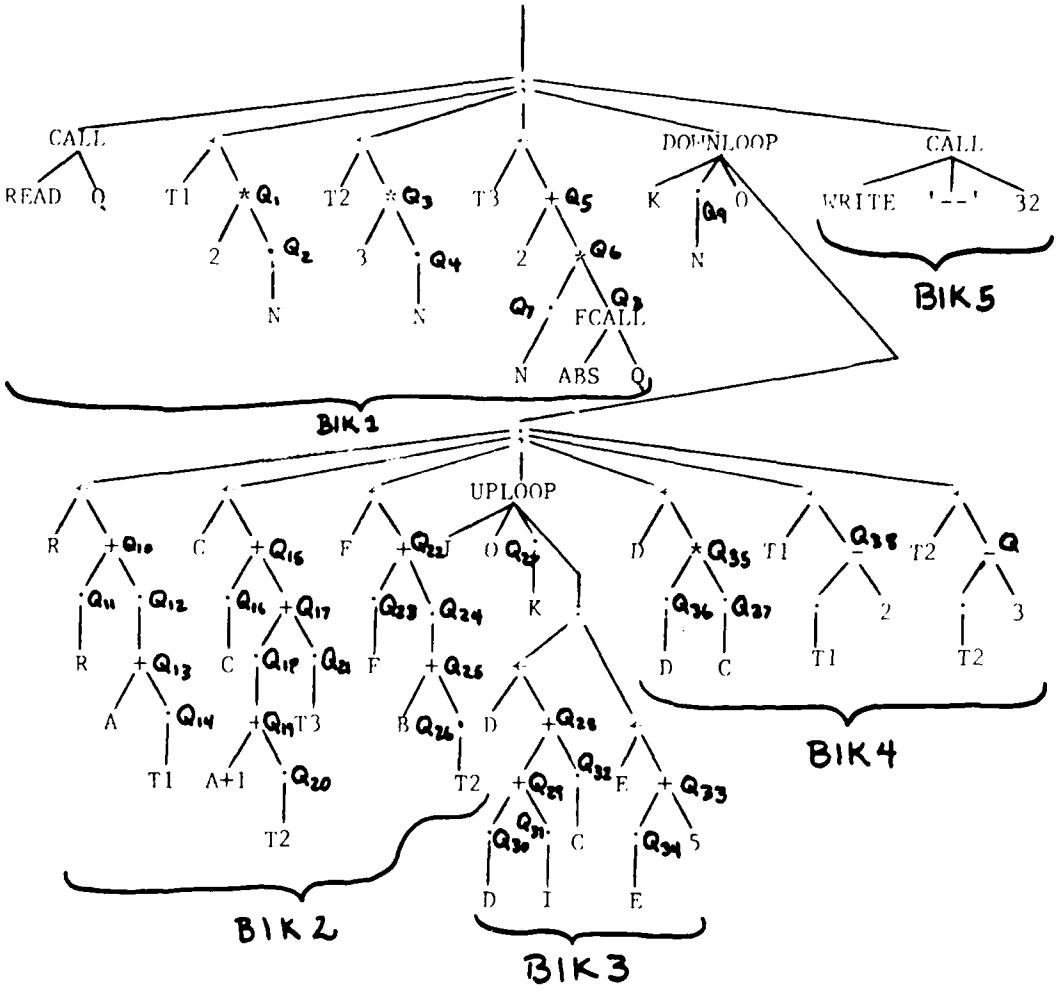


Figure 6.8 - Tree with Blocks and TNs Assigned

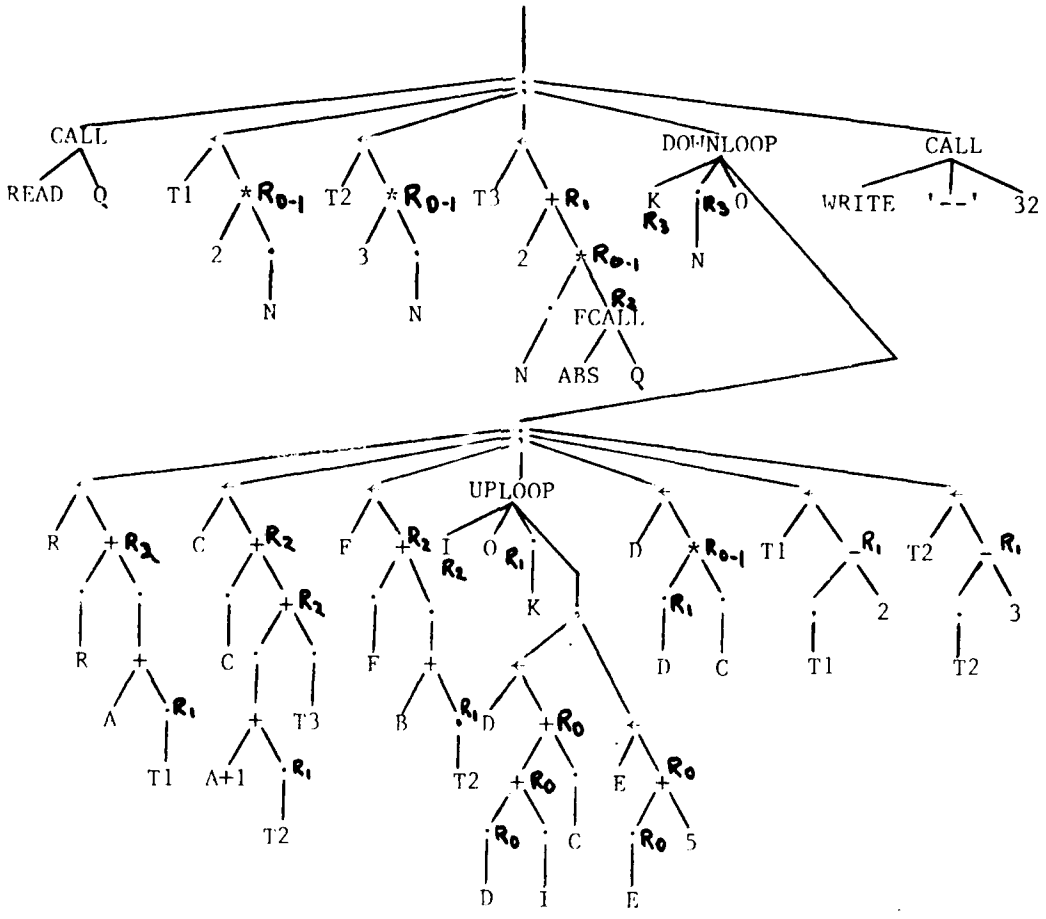


Figure 6.9 - Tree after Register Allocation

```
(← $1:REG 0) (EMIT [CLR $1])
(← $1:REG $2:REG)  (EMIT [MVR $1, $2])
(← $1:REG $2:MA) (EMIT [LD $1, $2])
(← $1:MA $2:REG) (EMIT [STA $1, $2])
(← $1:REG (+ $1:REG 1)) (EMIT [INC $1])
(← $1:REG (+ $1:REG $2:#16)) (EMIT [ADI $1, $2])
(← $1:REG (+ $1:REG $2:MA)) (EMIT [ADD $1, $2])
(← $1:REG (- $1:REG 1)) (EMIT [DCR $1])
(← $1:REG (- $1:REG $2:#16)) (EMIT [ADI $1, - $2])
(← $1:REG (* $1:REG 2)) (EMIT [SLA $1])
(← $1:REGP (* $1:REGO $2:MA)) (EMIT [MPY $1, $2])
(CALL $1:NAME $2:LIST) (EMIT [JSR $1 ; $2])
(← $1:REG2 (FCALL $1:NAME $2:LIST)) (EMIT [JSR $1 ; $2])
(UPLOOP $1:REG $2:MA $3:MA $4:TREE)
                        (EMIT [LDA $1, $2 ; GENLBL $5 ;
                         CPR $1, $3 ; BGR $6 ; $4;
                         INC $1 ; BRU $5 ; GENLBL $6])
(DOWNLOOP $1:REG $2:MA $3:MA $4:TREE)
                        (EMIT [LDA $1, $2 ; GENLBL $5 ;
                         CPR $1, $3 ; BLS $6 ; $4 ; DCR $1 ;
                         BRU $5 ; GENLBL $6])
```

Figure 6.10 Example LOP Description

matched immediately; the generated code is temporarily linked to the tree for later collection and the algorithm proceeds to the assignment of 2 * N T1. This is a little harder, since no tree directly matches the node (T1 is a memory location). The search algorithm will match the SLA instruction and then attempt to recurse on the rest, finding the STR instruction. Note that, because of the ordering of the LOP, the SLA was generated instead of an MPY instruction.

The algorithm will proceed through the rest of the tree in a similar manner. Finally, the algorithm will walk the tree, building the doubly linked list of instructions in correct order (Figure 6.11).

After code has been generated, FOP performs post optimization on the code. Because of the simplicity of this example, cross-jumping, simplification of operators, and span-dependent instruction optimization are not performed. The single post optimization done is the elimination of the redundant operation "LD R1, T2", which is marked with an asterisk.

```
        JSR   READ
        QLIST
        LD    R1, N
        SLA   R1
        ST    T1, R1
        LD    R1, N
        MPY   R1, THREE
        ST    T2, R1
        JSR   ABS
        QLIST
        MVR   R3, R2
        MPY   R3, N
        ADI   R3, 2
        ST    R3, T3
        LD    R3, N     # OPENING CODE
                        # FOR DOWNLOOP
L1:
        CPR   R3, ZERO
        BLS   L2
        LD    R1, T1
        LD    R2, R
        ADD   R2, A(R1)
        ADD   R1, C
        ST    R2, C
       *LD    R1, T2
        LD    R2, F
        ADD   R2, B (R1)
        ST    R2, F
        CLR   R2    # OPENING CODE FOR
L3:     CPR   R2, K    # UPLOOP
        BGR   L4
        LD    RO, D
        ADD   RO, I
        ADD   RO, C
        ST    RO, D
        LD    RO, E
        ADI   RO, 5
        ST    RO, E
        INC   R2    # CLOSING CODE FOR
        BRU   L3    # UPLOOP
L4:     LD    R1, D
        MPY   R1, C
        ST    R1, D
        LD    R1, T1
        ADI   R1, - 2
        ST    R1, T1
        LD    R1, T2
        ADI   R1, - 3
        ST    R1, T2
        DCR   R3    # CLOSING CODE FOR
        BRU   L1    # DOWNLOOP
L2:     JSR   WRITE
        WLIST
        .
        .
        .
ZERO    DATA 0
THREE   DATA 3
QLIST   DATA 1
        DATA Q
WLIST   DATA 2
        DATA '---'
        DATA 32
```

Figure 6.11 Code Generator Output

7. BIBLIOGRAPHY

This bibliograph is separated into sections based on content.

Computer Description Languages

General and Survey Material

Bar75 Barbacci, M. R.: "A comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Transactions of Computers, 2412 (February 1975), pp137-150.

SJ78 Smith, B. J. and Jordan, H. F.: "Implications of Series-Parallel Sequencing Rules", Computing 19 (1978) pp189-201.

Jor Jordan, H. F.: "Structural and Procedural Iteration in Hardware Description Languages", Arbeitsberiche des IMMD, Band 9, Heft 8, pp277-303 .

JS Jordan, Harry F. and Smith, Burton, J.: "Structure of Digital System Description Languages".

JS77 Jordan, Harry F. and Smith, Burton, J.: "The Assignment Statement in Hardware Description Languages", Computer, X, No. 6 (June 1977) pp43-49.

Su74 Su, Stephen Y. H. "A Survey of Computer Hardware Description Languages in the U.S.A.", Computer, VII, No. 17 (Dec. 12), pp45-51.

ISP (and derivatives):

BN71 Bell, C. G. and Newell, A.: Computer Structures: Readings and Examples, McGraw-Hill, 1971.

Sie74 Siewiorek, Dan: "Introducing ISP", Computer VII, No. 12 (Dec. 1974), pp39-41.

Bar79 Barbacci, M. R.: Instruction Set Processor Specifications (ISPS): The Notation and Applications, Dept. of Computer Science, Carnegie-Mellon University (May 1979) CMU-CS-79-123.

APDL:

Dar68 Darringer, J. A." A Language for the Description of Digital Computer Processors", Proceedings of the Design Automation Workshop, 1968,pp15-1 to 15-8.

PD67 Parnas, D. L. and Darringer, J. A.: "SODAS and a Methodology for System Design", Proceedings of the 1967 Fall Joint Computing Conference, pp449-474.

LALSD:

SB75a Su, S. Y. H. and Baray, M. B.: "Logic/System Design Automation Part I: LALSD - A Language for Automated Logic and System Design", IEEE Computer Society Repository, 1975.

SB75b Su, S. Y. H. and Baray, M. B.: "LALSD - A Language for Automated Logic and System Design" 1975 Int'l Symposium on CHDL's and Their Applications Proceedings, pp30-31 ACM, New York, 1975.

BS71a Baray, M. B. and Su, S. Y. H.: "A Digital System Modeling and Design Language", Proceedings of the 8th Annual Design Automation Workshop 1971, pp1-22.

BS71b Baray, Mehmet B. and Su, Stephen Y. H.: "A Digital System Modeling Philosophy and Design Language" Proc. of Design Automation Workshop SHARE - ACM - IEEE, June 1971, pp1-14.

SMITE:

TRW77 TRW Defense and Space Systems Group: SMITE Reference Manual, (November 1977), RADC-TR-77-364. (A049038)

LCD:

IBM IBM: LCD - Language for Computer Design, Language Reference Manual.

AHPL:

HP73 Hill, F. J. and Peterson, G. R.: Digital Systems: Hardware Organization and Design, Wiley, New York, 1973.

Hil74 Hill, Frederick F.: "Introducing AHPL" Computer, VII, No. 12 (Dec., 1974), pp28-30.

Hil75 Hill, F. J.: "Updating AHPL", 1975 Int'l Symposium on CHDL's and Their Applications Proceedings, ACM, New York, 1975, pp22-29.

APL:

Ive72 Iverson, K. E.: "A Common Language Hardware, Software and Applications", Proceedings of the 1972 Fall Joint Computing Conference, pp121-129

Ive63 Iverson, K. E.: "Programming Notation in Systems Design" IBM Systems Journal, (June 1963) pp117-129.

Fri67 Friedman, T. D.: "ALERT: A Program to Compile Designs from New Computers", Digest 1st Annual IEEE Computer Conference, (Sept. 1967) pp128-139.

FY69 Friedman, Theodore D., and Yank, Sih-Chin,: "Methods Used in An Automatic Logic Design Generator (ALERT)", IEEE Transactions on Computers, C-18, No. 7, (July, 1969), pp593-606.

CDL:

Chu65 Chu, Yaohan: "An ALGOL-like Computer Design Language", Communications of the ACM, VIII, No. 10 (Oct. 1965) pp607-615.

Chu74 Chu, Yaohan: "Introducing CDL", Computer, VII, No. 12 (Dec., 1974), pp31-33.

DDL:

DD68 Duley, James R. and Dietmeyer, Donald L.: " Digital System Design Language (DDL)", IEEE Transactions on Computers, Vol. C-17, No. 9 (Sept. 1968) pp850-861.

Die74 Dietmeyer, D. L.: "Introducing DDL", Computer, VII No. 12 (Dec., 1974), pp34-38.

Cassandre:

BGL71 Bogo, Cayot, Lux, Mermet, and Payan: "CASSANDRE and the Computer Aided Logical Systems Design", TA6, 26, Proceedings of IFIP Congress, 1971.

Hof Hoffman, R.: "Experiences with the Language Cassandre".

ERES:

GHH77 Gordill, R., Handler, W., HeBlins, H., Klar, R., Spies, P. P.: ERES:A Nonprocedural Computer Hardware Design Language with Precise Description of Timing, Universitaten Bonn und Erlangen, Band 10, Nummer 15, Erlangen, November 1977.

GJK Gardill, R., Jordan, H. F. and Klar, R.: " CHDL for Description of Semisynchronous Networks".

LOGAL:

Ste77 Stewart, J. H.: "LOGAL: A CHDL for Logic Design and Synthesis of Computers", Computer, X, No. 6 (June, 1977) pp18-26.

Extended CDL:

MS75 Mowle, F. J. and Stine, L. R.: Reference Manual for Purdue Extended CDL Version 4.2, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, TR-EE 75-15, 1975.

LOTIS:

Sch64 Schlaeppi, H. P.: "A formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)", IEEE Transactions on Electronic Computers EC-13 (August 1964), pp439-448.

ConLan:

Pil77 Piloty, Robert: Memo 3.20 Progress Report of the Working Group of the Conference on Computer Hardware Description Languages, Conference on Digital Hardware Languages, University of Texas at Austin, June, 1978.

RTS III:

Pil75 Piloty, Robert: "Segmentation Constructs for RTS III, a Computer Hardware Description Language Based on CDL", Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and their Applications, IEEE Catalog #75CH1010-8C (1975) pp115-124.

Compilers and TWSs in General

ACH77 Allen, F. E., Carter, J. L., Harrison, W. H., et al.: "The Experimental Compiling System Project", Research Report RC 6718 (#28922) IBM Thomas J. Watson Research Center, (Sept. 1977).

AU73 Aho, A. V., and Ullman, J. D.: The Theory of Parsing, Translating, and Compiling, Prentice-Hall, 1973.

AU77 Aho, A. V. and Ullman, J. D.: Principles of Compiler Design, Addison-Wesley, 1977.

BHH77 Buckles, B. P., Hodges, B. C. and Hsia, P.: A Survey of Compiler Development Aids, NASA-TM-X3490 (February 1977).

BW75 Bochmann, G. V. and Ward, P.: "Compiler Writing System for Attribute Grammars", Department D'Informatique, Universite de Montreal, (July 1975)

Cro72 Crowe, David: "Generating Parsers for Afix Grammars" Communications of the ACM, 15, 8, (Aug. 1972)

Dun75 Dunbar, Terry L.: "JOCIT JOVIAL Compiler Implementation Tool", Computer Sciences Corporation, RADC-TR-74-322, (Jan. 1975).(A005307)

FG68 Feldman, J. and Gries, D.: "Translator Writing Systems", Communications of the ACM 11, 2, (Feb. 1968).

II74 Intermetrics Inc.: "HAL/S-FC Compiler System Functional Specification", IR #59-4, (July 1974).

Knu68 Knuth, D. E.: "Semantics of Context - Free Languages", Mathematical Systems Theory, 2, 2, (1968)

Knu74 Knuth, D. E.: "Structured Programming with GO TO Statements", Computing Surveys 614 (December 1974), p. 268.

LCH79 Leverett, Bruce W., Cattell, Roderic G. G., Hobbs, Steven O., et al.: "An Overview of the Production Quality Compiler-Compiler Project", Computer Science Department, Carnegie-Mellon University, (Feb. 1979), CMU-CS-79-105

LRS74 Lewis, P. M., Rosencrantz, D. J. and Stearns, R. E.: "Attributed Transformations", J. of Computer and Systems Sciences, 9, (1974).

LS76 Lancaster, Ronald L. and Scheider, Victor B.: "Quick Compiler Construction Using Uniform Code Generators", Software - P&E, 6 (1976) pp83-91.

RFP Rome Air Development Center: Request for Proposal: Retargetable Compiler, 1978.

WJW75 Wulf, William, Johnsson, Richard K., Weinstock, Charles B., et al.: "The Design of an Optimizing Compiler", American Elsevier, (1975).

Intermediate Languages

Col74 Coleman, Samuel S.: "JANUS: A Universal Intermediate Language", (PhD Thesis), University of Colorado, PB-232-923, (May 1974).

CPW74 Coleman, S. S., Poole, P. C. and Waite, W. M.: "The Mobile Programming System, JANUS", Software - P&E, 4 (1974) pp5-23.

Fra79 Frailey, Dennis J.: "An Intermediate Language for Source and Target Independent Code Optimization", SIGPLAN Notices 14, 8, (Aug. 1979).

HW78 Haddon, B. K. and Waite, W. M.: "Experience with the Universal Intermediate Programming Language JANUS", Software - P&E 8, 5 (September - October 1978), p. 601.

MOS58 Mock, O., Olsztyn, V., Strong, J., Steel, T., Tritter, A. and Wegstein, J.: "The Problem of Programming Communications with Changing Machines: A Proposed Solution", Communication of the ACM, 1, 12, (August 1958)

Ric71 Richards, M.: "The portability of the BCPL Compiler", Software P&E, 1, 2 (1971) pp135-146.

SLN79 Schatz, B. R., Leverett, B. W., Newcomer, J. M., et al.: "TCOL Ada: An Intermediate Representation for the DOD Standard Programming Language", Report CMU-CS-79-42, Computer Science Department, Carnegie-Mellon University, (March 1979).

WH78 Waite, W. M. and Haddon, B. K.: "The Universal Intermediate Language JANUS (Draft Definition)", Report SEG-78-3, Department of Electrical Engineering, University of Colorado, (Sept. 1978).

Optimizations

CLI75 Computer Linguistics, Inc.: "A survey of Optimization Techniques in Compilers", RADC Report, (Sept. 1975).

Fra79 Fraser, C. W.: "A Compact, Machine-Independent Peephole Optimizer", Sixth Annual ACM Symposium on Principles of Programming Languages, (Jan. 1979), pp 1-6.

Ges72 Geschke, C. M.: Global Program Optimization, Dept. of Computer Science, Carnegie-Mellon University (October 1972), AFOSR-TR-73-1059.

HS77 Hunt, J. W. and Syzmanski, T. G.: "Gast Algorithm for Computing Longest Common Subsequences", Communications of the ACM, 20, 5, (May 1977).

LM69 Lowry, Edward, S., and Medlock, C. W.: "Object Code Optimization", Communications of the ACM, 12, 1 (Jan. 1969) p. 13.

McK65 McKeeman, W. M.: "Peephole Optimization", Communications of the ACM 8, 7, (July 1965).

Szy78 Szymanski, Thomas G : "Assembling Code for Machines with Span-Dependent Instructions", Communications of the ACM 21, 3 (April 1978).

Code Generation and Register Allocation

AJ76 Aho, A. V. and Johnson, S. C.: "Optimal Code Generation for Expression Trees", Journal of the ACM, 23, 3, (July 1976), pp488-501.

AJ77 Aho, A. V. and Johnson, S. C.: "Code Generation for Expressions With Common Subexpressions", Journal of the ACM, 24, 1, (January 1977) pp146-160.

Amm77 Ammann, Urs: "On Code Generation in a PASCAL Compiler", Software - P&E 7, (1977), pp391-423.

Bea Beatty, J. C.: "Register Assignment Algorithm for Generation of Highly Optimized Object Code", IBM J. Res. Develop., (Jan. 1974).

BS76 Bruno, John and Sethi, Ravi: "Code Generation for a One-Register Machine", Journal of the ACM, 23, 3, (July 1976) pp506-510.

Car77 Carter, J. Lawrence: "A Case Study of a New Code Generation Technique for Compilers", Communications of the ACM, 20,12, (December 1977).

Cat77 Cattell, R. G. G.: "A Survey and Critique of Some Models of Code Generation", Carnegie-Mellon University, Department of Computer Science, (Nov. 1977).

Cat76 Cattell, R. G. G.: "Describing Machine Data Types in a Machine Description Language", Department of Computer Science, (August 1976).

Cat78 Cattell, R. G. G.: "Formalization and Automatic Derivation of Code Generators", (PhD Thesis), Carnegie-Mellon University, Department of Computer Science, (April 1978) CMU-CS-78-115.

CG77 Crenshaw, J. W. and Griffin, D. R.: "HOL Code Generator Development Methodology", Computer Sciences Corporation, (Nov. 1977).

CHM78 Chu, E., Halb, E., McCoy, H. and Morton, R.: "Automated Code Generators for Compilers", RADC-TR-78-157, (Aug. 1978).(A059699)

DNF79 Donnegan, Michael, K., Noonan, Robert E., and Feyock, Stefan: "A Code Generator Language", SIGPLAN Notices 14, 8, (Aug. 1979).

ER70 Elson, M. and Rake, S. T.: "Code - Generation Technique for Large-Language Compilers", IBM Systems Journal 3, (1970).

Fra77 Fraser, Christopher W.: "Automatic Generation of Code Generators", (Thesis), University of Arizona, Department of Computer Science, (July 1977).

Fre74 Freiburghouse, R. A.: "Register Allocation Via Usage Counts", Communications of the ACM 17, 11, (Nov. 1974).

Gla73 Glanville, Robert S.: "A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers", (PhD Thesis), University of California, Berkeley, (Dec. 1973).

Joh75 Johnsson, R. K.: An Approach to Global Register Allocation, Dept. of Computer Science, Carnegie-Mellon University (December 1975) AFOSR-TR-76-0603.

New75 Newcomer, Joseph M.: "Machine-Independent Generation of Optimal Local Code" (PhD Thesis), Department of Computer Science, Carnegie-Mellon University, (May 1975).

Noo79 Noonan, Robert E.: "The Design of Relatively Machine-Independent Code Generators", NASA Contractor Report 159016, (Feb. 1979).

Oak79 Oakley, John D.: Symbolic Execution of Formal Machine Descriptions (PhD Thesis), Dept. of Computer Science, Carnegie-Mellon University (April 1979), CMU-CS-79-117.

Sit79 Sites, Richard L.: "Machine-Independent Register Allocation", SIGPLAN Notices 14, 8, (Aug. 1979).

Ter78 Terman, Christopher J.: "The Specification of Code Generation Algorithms", (Thesis), MIT, (Jan. 1978).

Wei73 Weingart, Steven W.: "An Efficient and Systematic Method of Compiler Code Generation", (PhD Thesis), Yale University, (1973).

Appendix A:

A Comparative Study of Computer Descriptions Languages

This report was completed under the auspices of the Martin Marietta Independent Research and Development program. Report # R79 - 48659 - 001

Contents

Introduction

This paper briefly describes a number of computer description languages (CDL's). Rather than give a separate description of each language, the discussion focuses on groups of characteristics. The languages are described and compared in terms of each group. This should serve to make the presentation easier to comprehend and allow the reader to select those aspects of CDL's of most interest to him. Section 2 presents the procedural languages, while Section 3 presents the non-procedural languages. Each language group is summarized separately.

Because of the large number of existing CDL's, this paper does not try to be exhaustive, though an effort was made to include languages with wide applicability. Section 4 gives some of the languages that were left out and some justification for doing so. Section 5 discusses two languages, ConLan and RTS III, that look promising and ambitious, but which are still being defined. Section 6 gives some conclusions and summary remarks. Section 7 summarizes the languages in a table for easy reference and comparison.

Procedural Languages

The distinguishing feature of a procedural language is that it attributes a significance to the lexical order of the action statements. Generally, the action statements are grouped into "steps" (or "time blocks"). The statements within a given group are assumed to operate in parallel. The groups are performed in order as they are listed in the text. Generally, different sequences of groups can operate in parallel. This series-parallel sequencing structure can generally be nested to any depth. Some languages have more general parallelism constructs.

The languages considered here are ISP [bN71], APOL [Barb8], LALSD [SB75a], SMITE [TRW77], and LCU [Ib^4]. The languages are described and compared on the basis of a few characteristics at a time.

Purpose, Description Type and Level of Description

First, the languages are compared on the basis of purpose, description type, and level of description.

Purpose here refers to the applications a language is targeted for. A language designer may focus on goals for a language that are not directly related to the target applications (readability, writability, careful treatement of time, powerful composition constructs, extensibility, etc.). Many of the areas addressed by these goals are discussed later. Others are too sub-

jective for systematic evaluation and comparison. Thus, someone interested in a comprehensive treatment of the objectives of a language is best advised to refer to the language defining document.

Description type refers to the general aspects of the ways a language can be used to describe a digital system. First, a description may emphasize the behavior of a system (the actions performed by the system) or its structure (the components and their relationships). While pursuing either of these orientations or some combination, a description may give a specification, treating the system as a "black box" and giving only its I/O behavior. Alternatively, a description may give an implementation, minutely detailing the behavior and/or structure of the system. It should be noted that a language can allow, or insist on, redundancy in the form of alternative descriptions for a system or subsystem. The alternatives may have different orientations and/or different degrees of detail. This redundancy may be exploited through human or machine consistency checking. To summarize, description type includes the orientation (structure vs. behavior), level of detail (specification to implemtation), and redundancy possible with a language.

The level of a description refers to the level of primitives used in writing the description. The level may be component circuit, switching circuit, register transfer, ISP, or PMS. The component circuit level deals with diodes, transistors, resistors, etc., and their interconnections. The switching circuit level deals with logic gates and flip-flops. Register transfer primitives include registers, combinatorial expressions and dis-

crete data and control operation steps. The ISP (instruction set processor) primitives are interpretation rules, the memories visible to the programmer and instructions (in terms of visible memories). The PMS level describes the gross components of a system (Processors, Memories, Switches, devices, etc.) giving their general capabilities and their interrelationships. The reader should be careful to distinguish level of descriptiton from level of detail. The two do correlate weakly, but the latter is concerned with the kinds of primitives available for writing a description, while the former is concerned with how detailed a description is. For a more through discussion of description levels, the reader should see [BN71] and [Bar75].

ISP

The language ISP was developed for exposition. Particularly, its initial purpose was to concisely describe the instruction sets for various diverse computers. It is oriented toward describing behavior with some aspects of the structure being implied. ISP is useful on all levels of detail, from top-level specification to a detailed discription of an implementation. ISP does not allow for redundancy. ISP in mainly a register transfer language, though it can be used at the ISP level of description.

APDL

APDL was designed for design, simulation, and documentation. It is oriented toward behavioral descriptions. The descriptions are close to being a specification. APDL provides register transfer level primitives. Redundant descriptions are not allowed.

LALSD

LALSD was intended for use in documentation, simulation, and design. It is oriented toward describing the structure of a system. The description may be at any level of detail from specification to implementation. At any level of detail above implementation, a high-order language may be used to describe the behavior of some parts of a system. Redundant descriptions are not allowed. LALSD's primitives are on the register transfer level.

SMITE

SMITE was developed for use in developing emulators to run on a QM-1. SMITE is oriented toward behavioral diescriptions. It can be used for specifications through implementation descriptions. No redundancy is allowed. SMITE's primitives are on the register transfer level.

LCD

LCD is intended for use with computer design. It is oriented toward behavioral descriptions. It is not suited to writing high level specifications, but it can suppress some detail, though the language is mainly appropriate to describing specific implementations. Redundancy is required. Each module must have at least a general description of behavior. Each module, except one on the lowest level, must also have a description of the data objects and control sequences which implement its behavior. LCD is mainly a register transfer level language, but can be used as an ISP level language.

Data, Carriers, and Assignments

Now the languages are compared on the basis of the data facilities, carriers and assignments they provide. Data supported for compilation and in the described machine are distinguished. Carriers are the elements of a system that hold and transmit data. They may simply be terminals (wires or connection points) which retain a given value only so long as that value is applied to them. Busses, which might be considered a variety of terminal, sometimes recieve special support from languages. This and the fact that they are so widely used merit busses being singled out as a carrier type when a language supports them. Finally, a carrier may be a register, an element that retains a value over time without an input being applied.

Assignments may serve several purposes in a computer description. They may represent setting some register or applying an input to a terminal. Assignments may set the value of compile-time bookkeeping variables. Assignment may be used to represent multiplexing or pulsing. Finally, assignment may represent renaming of some structure. Each one of these functions may have a separate symbol. For a thorough discussion of the assignment statement, see [JS77].

ISP

ISP supports bits and integers as data both during compilation and in the described machine. Both constants and variables are supported. Vectors and matrices of bits are supported. A vector of bits or a row of a matrix may be used as an integer. Data format description tools have never been defined for ISP. Indexing and renaming are supported for accessing. An index expression may specify a range or list of indexes. Any data object may appear in an index expression. Any data object, indexed or not, or collection of them, may be given a name. The renaming may also be used to view the object or collection as an array or matrix with index ranges of its own. Thus, names may be given to subregisters and collections of registers. ISP primitive operations include the normal arithmetic operators, logical AND, OR, exclusive OR, equivalence and NOT. The primitive operators also include all the relational operators and a rich set of shift operators. Expressions may be arbitrarily complex. The only carriers ISP supports are registers. Renaming and transfer assign-

ments are supported.

APDL

APDL provides binary, octal, and decimal data objects. Also supported are switches, data objects taking on statement labels as values. The numeric objects may be composed into vectors and matrices. Matrix rows and vectors may be treated as positive integers. No description of abstract formats is possible. The indexing and renaming capabilities are analogous to ISP, except that an index for referencing a row of a matrix may only specify a single row.

The boolean operators, AND, OR, NOT, and exclusive OR, are provided. They treat 0 as true and 1 as false. The standard arithmetic operators are provided for integers, and all the standard relational operators are provided. Darringer's article [Dar68] does not specify any expression composition, but does seem to allow their inclusion in a language implementation. Registers are the only carriers supported, and transfer is the only assignment supported.

LALSD

The only data type supported by LALSD is bits. Bits may be composed into vectors. Collections of objects and vectors can be renamed as vectors. Explicit address registers must be given for "memories", arrays of multi-bit registers. Indexing of memories must use this address register. Primitive operations include in-

crement, decrement, shift, complement and concatenation for vectors, treating vectors as integers when appropriate. AND, OR, NOT and exclusive OR are provided for bits. The full range of relational operators are available which treat vectors as integers. Conditions may be arbitrarily complex combinations of bit and relational operations.

Registers and terminals are supported as carriers. Assignments can be used for connection. Transfer is represented with a command that looks like a proceedure call.

SMITE

SMITE provides bits as a primitive data type. They can be composed into vectors and matrices. Formats may be described. They give names to subwords of abstract structures. These named subwords can then be used to reference subwords of concrete objects. Also, specific sections of specific objects can be given unique names. Any value can be used as an index. Primitive operations include addition, subtraction, a full complement of relational operators, AND, OR, NOT, exclusive OR, and concatenation. These operations, together with assignment, may be composed arbitrarily into expressions, so long as each subexpression returns a single value. Assignment returns the value assigned. Operations are evaluated from right to left normally, but parentheses may be used to overide the usual interpretation.

Only register type carriers are supported, and transfer is the only assignment supported.

LCD

LCD provides support for bits. A bit may take on a value of 0, 1, or UNDEFINED. LCD also supports variables that take on symbolic values. These variables may be tested for equality or inequality only. This support is helpful for simulation using symbolic execution. Vectors and matrices of bits may be specified. Concatenation, AND, OR, NOT, and the standard arithmetic operators are provided. The arithmetic operators treat vectors and rows of matrices as integers. AND reduction of vectors is also provided. Symbolic manipulation of symbolic values is provided. Any expression may be used as an index. Expressions may be arbitrarily complex.

Busses, registers, and terminals are supported as carriers. Assignment to a register is a transfer. Assignment to a terminal maintains the assigned value in the target for a time step.

Modularization, Control and Its Relation to Data, and Time

This section deals with a set of characteristics of how a language describes the decomposition of structures and the decomposition and control of processes. These characteristics are: 1) the modularization concepts; 2) the control constructs; 3) the treatment of time; and 4) the relationship between control and data.

It is useful to modularize a system in both time and space. Space modularization is seen in the partitioning of a system into memories, controls, ALU's, and busses. Time modularization is

seen in the definition of instruction fetch cycles, interface protocols, and instruction decode processes.

A language must provide ways of describing the processes and structure that make up a machine. For nontrival processes, mechanisms must be provided for making choices and specifying iteration. Since computers make major use of parallelism, a language must be able to express parallelism and coordination of parallel processes. A description of hardware structures must include descriptions of their interconnections. Finally, languages must also provide forms for excressing modularization.

Connection, sequencing, iteration, decision, and modularization are more concerned with the organization of the actions and components of a machine than with what the actions and components actually are. Organizational forms and mechanisms are refered to as control constructs. The kinds of control constructs provided by a language greatly affect its power and ease of use. For instance, a series of if statements can be used to make a choice among several alternatives, but a CASE or DECODE statement for the same decision is much easier to write and clearer to understand. Macros can be very useful, as is the ablility to apply descision mechanisms to structure. The constructs then become compile-time mechanisms. Iteration can be a particularly powerful and clear way of describing a large regular structure. Then there are structure and process control constructs, which can be divided into sequencing and synchronization constructs. Sequencing constructs include selection and FORK. Synchronization constructs include JUIN and signal-wait.

Time is of great importance to any computer design. Languages provide various ways of measuring and describing timing.

A common characteristic of the languages described in this paper is that they provide for the separation of control and data. The data are the memories and terminals and the control is the processes. It is also important that the control and data be able to interact. It is convenient to save the control state as data. Control state changes must sometimes be based on data. Thus the possible interactions of control and data for a languages are discussed.

ISP

ISP provides for modularization in the form of subprocedures and parameterized structures. Thus, macros are provided. Sequencing is series-parallel in straight text. IF..THEN..ELSE and DECODE are provided for control flow selection. Recursion is the only iteration construct available. SIGNAL-WAIT synchronization is provided. Timing is virtually ignored in that one cannot specify the time requirements of processes and the only synchronization possible is through the series-parallel sequencing or explicit synchronization through SIGNAL and WAIT. Control and data are kept separate except that control decisions may depend on data through IF...THEN...ELSE... and DECODE constructs.

AD-A084 195 MARTIN MARIETTA AEROSPACE DENVER CO DENVER DIV F/G 9/2
RECOMMENDATIONS FOR A RETARGETABLE COMPILER.(U)
MAR 80 G E HEYLIGER, L L MCELHANEY, T H DWYER F30602-78-C-0297
UNCLASSIFIED MCR-79-631 RADC-TR-79351 NL

2 OF 2
AD
A084195

END
DATE FILMED
6-80
DTIC

APDL

APDL provides process modularization through procedures and functions. IF..THEN..ELSE is supported as a sequencing construct. So are GOTO and IF EVER, which is similar to the ON CONDITION of PL/1. Another aid to decribing sequencing is the time block. The statements inside a time block are executed in parallel. The time blocks may not contain time blocks. Time requirements for time blocks may be specified. The data type SWITCH from ALGOL is available, and may hold an array of labels as values. It can then be used to facilitate a sequence selection.

LALSD

Structural modularization is supported by UNITs and FUNCTIONs. UNITs describe related structures and control processes, while FUNCTIONs describe combinatorial networks used within a UNIT. UNITs may be nested. Statements in a UNIT may use objects defined in an outer unit through use of an import specification. Sequencing constructs similar to IF..THEN..ELSE, CASE, and FORK..JOIN are provided. Iteration on a condition can be specified. It is possible to wait on a condition or have execution of a UNIT be dependent on the setting of some control variables. Control and data are very strictly separated, though control decisions may be based on data values. It is possible to specify one or more clocks, which may be independent or synchronized with other clocks.

SMITE

SMITE provides for subprocedures through PROCESSes. IF..THEN..ELSE and CASE are provided for sequence selection. Iteration on a condition and for a count are provided. A DO FOREVER construct is provided, and a loop ESCAPE. PARALLEL-BEGIN and PARALLEL-END are provided to support series-parallel sequencing. The time requirement for any simple or composite operation may be specified. Control and data are carefully separated, but control decisions may be based on data values.

LCD

LCD supports subprocedures. It has block structure scope rules. IF..THEN..ELSE is provided for sequence selection and WHILE is provided for procedural iteration. One implicit, discrete clock is assumed. Relative timings of events may be specified based on this clock. These timings can override lexical order in specifying sequence. Control and data are kept separate.

Generality, Readability and Writability

The languages are now compared on the basis of their generality, readability and writability. These characteristics are similar to those discussed by Barbacci [bar75]. The main points affecting generality that are examined are assumptions about the machines being described, whether the language can work on dif-

ferent description levels, extensibility, existing applications based on the language and machines described in the language. Of these five points, it is hardest to get good information on what machines have been described in a language. The lists given for described machines should not be taken as at all exhaustive, but just as miscellaneous information. The aspects of writability and readability looked at are familiarity of language structures and operators, simplicity and fidelity possible between the structure of a description and the described machine.

ISP

ISP has shown itself to be capable of describing a wide variety of machines. Bell and Newell give numerous descriptions in ISP [BN71]. In addition, Carnegie-Mellon University has compiled numerous descriptions in ISPS, a variant of ISP. Among the described machines are the IBM 360, IBM 370, Mark I, PDP8, PDP10, and various PDP11's. One reason for this generality is that ISP makes few assumptions about machines. The main ones are that the state of a machine is embodied in its memories and that all module interfaces are realized with registers. ISP is mainly suited to the register transfer level of description. It can come close to the ISP level. By restricting the operators and the complexity of expressions, ISP can be stretched to the switching circuit level. ISP has explicitly allowed for extensions through allowing qualifiers on all its expressions, operators and constructs. These qualifiers can be used to put extra information into a description. The language can also be extended by adding

new operators or new attributes for modules and registers. Furthermore, information required by special automated uses of an ISP description may be put into comments. ISP descriptions have been translated into low level computer fabrication instructions and parts lists. They have been the basis of automated software generation and simulations. ISP has also been used for classroom design efforts.

ISP is highly readable and writable, since, after all, it was originally developed for exposition. Its structures and operators are largely based on ALGOL. One indication of its simplicity is the small size (6-8 pages) of its own BNF grammar description. It is therefore simple and familiar. Its general modularization support and support of vectors and matrices of bits allow a description to closely follow the structure of the described machine. This fidelity is also enhanced by the ability to share resources between modules and specify explicit synchronization and series-parallel sequencing.

APDL

APDL makes no explicit assumptions about the machines it describes. It can be used on an ISP as well as a register transfer level. It is extensible through new operators. It has been used for simulation and to generate detailed hardware descriptions. No data has been found on what devices have been described in APDL.

APDL closely resembles ALGOL and has most of the simplicity and familiarity of that language. Its general modularization ca-

pability should allow close fidelity between a description and the described machine.

LALSD

In LALSD a digital system consists of a collection of units whose operation is controlled through explicit control signals. Each unit in turn has the ability to generate sequences of control signals to operate its subcomponents. The language can only work at the register transfer level. It is extensible through the addition of new operators. It has been used for simulation and logic description generation.

LALSD is a rather complex language requiring all descriptions to have a fairly complex structure. It would have a hard time describing with much fidelity a machine that was not a system of cooperating units, such as a machine implemented through micro-programming, where the complexity lies in the stored control program more than in the hardware. While LALSD provides a lot of familiar and useful facilities, it uses unfamiliar constructs to support them. For instance, "/ condition /" means "wait on condition" and " [A] => B" means "IF A THEN B".

SMITE

SMITE is only useful at a register transfer level of description. It makes no restrictive assumptions about the machines it describes. It is extensibile through the addition of operators. SMITE is used for writing emulators for the DM-1.

The language is simple, and uses familiar constructs. Its modularization capabilities should allow a description to closely follow the structure of the described machine.

LCD

LCD assumes that the operations in a described machine take place in discrete steps in time with a single system clock. It is useful at the register transfer level only. It can be extended through introducing new operators. LCD is being used for design and simulation.

The language uses familiar constructs and allows for simple descriptions except for the requirement for redundancy. The redundancy may aid understanding of the description and automated consistency checking. The modularization capabilities of the language should allow the structure of a description to closely follow the structure of the described machine.

Summary

It can be seen that there is a great deal of similarity among the various procedural languages, especially in the facilities they provide. Puposes vary, but most are oriented toward describing the behavior of a system at any level of detail using RT primitives. Data objects supported are similar among the languages, with bits and integers being provided along with vectors and matrices of bits. Integers are assumed to be vectors of

bits. While only SMITE provides for abstract format specifications, most languages allow the renaming of fields of specific vectors for ease of access. Indexing is always supported, but sometimes restricted to use with explicit address registers or simple expressions. Arithmetic, boolean, and relational operators are generally provided along with concatenation and shift operators. All of these operators are available for use in arbitrarily complex expressions. Registers and terminals are usually provided as data carriers along with transfer and connection assigments. Some concept similar to subprocedures and functions is usually provided for modularization, though the form of the control constructs and their exact semantics do vary. Most of the languages are rather easy to write and understand, since they all (to some degree) are simple, use familiar constructs, and provide a fair degree of fidelity to the described hardware. Also, a wide variety of control constructs are provided. However, IF..THEN..ELSE.., or something analagous, is the only construct always provided. Generality and treatment of time very widely among languages.

ISP, with its unique emphasis on exposition, is the most general of the procedural languages and is the most extensible, as well. Its series-parallel sequencing and signal-wait synchronization primitives are very high level, lacking the ability to precisely describe fine timings.

APDL is unique in being mainly concerned with specification of a system without attention to design details. In support of this, APDL provides the widest variety of data types, including state encodings (called switches), which are not supported by any

other procedural languages. In addition, APDL is the only procedural language to have a GOTO or an interrupt handling primitive (IF EVER...). Its time block, which allow for some parallelism and timing specification, is unusual, also.

LALSD is the only structurally oriented procedural language and is also, appropriately, the only procedural language to feature a connection assignment. LALSD is a lower level language than the other procedural languages, as exemplified by its reliance on metronome-like clocks and the requirement for an explicit address register. One interesting aspect of LALSD is its use of controlled scopes. Otherwise, however, it is less easy to use than the other languages due to its complexity and its use of unfamiliar constructs.

SMITE is the only language designed for emulation. Otherwise it is distinguished by its wide variety of useful and familiar control constructs and its ability to precisely describe timing requirements.

LCD is the only language to allow for redundant descriptions. It also allows for precise description of timing.

Non-procedural Languages

The languages described in this section are CDL [Chu65], DDL [DD68], MOP [Cat78], Cassandre [BGL71], ERES [GHH77], and AHPL [HP73], [Hil75]. The organization of the discussion in this chapter will be similar to that of the second chapter. Characteristics of the languages will be grouped in the same way with the languages being compared on the basis of each group in turn. The definition of terms and discussion of issues will not be repeated. The reader is left to go to the second chapter for this material.

Purpose, Description Type, and Level of Description

CDL

CDL was developed for use with digital design and simulation. Descriptions in CDL specify a very detailed implementation. These descriptions are oriented toward the behavior of a system rather than its structure. CDL primitives are on the register transfer level. Redundant descriptions are not allowed.

DDL

DDL was developed for digital design. It can be used for a wide range of levels of detail. The descriptions are oriented

toward the behavior of a system. DDL primitives are at the register transfer level of description. Redundant descriptions are not allowed.

MOP

MOP was intended for use with the automatic generation of software. It is oriented toward the behavior of machines and the descriptions written in MOP are high level specifications. Redundant descriptions are not allowed. MOP's primitives are strictly on the ISP level of description.

Cassandre

Cassandre's purpose was to aid design and developement of digital systems. Cassandre is oriented toward describing the behavior of a system more than its structure. Descriptions in Cassandre can be at any level of detail from a high level specification to a detailed implementation. Cassandre primitives are at the register transfer level of description. There is no allowance for redundant descriptions.

ERES

ERES was designed for use in computer hardware design. ERES is oriented toward describing the structure of a computer, and can be used for descriptions over a wide range of levels of detail. Redundant descriptions are not allowed. ERES can be used at

the register transfer or switching circuit level of description.

AHPL

AHPL was developed for use in teaching design. Descriptions in AHPL are non-redundant and are oriented toward the behavior of a system. AHPL is appropriate for writing specifications and can be used to describe a fairly detailed specification. The language is mainly useful on the register transfer level of description, though it can be used to do some description on the switching circuit level.

Data, Carriers, and Assignments

CDL

CDL supports bits and integers in the described machine. Vectors of bits may be defined. Memories, arrays of register vectors, may be defined with an explicit address register for indexing. Vectors of bits may be treated as integers. Register subfields and collections of registers may be given names for ease of reference. Addition, subtraction, incrementation and decrementation are defined for integers. The logical operators AND, OR, NOT, and EQUIVALENCE are provided. A full set of shift and relational operators are provided. A concatenation operator is provided. In addtion, FETCH and STORE are provided for memories. Since CDL operations are supposed to be done in a single

clock period, expressions must be kept simple.

Register and terminal carriers are provided, along with separate transfer and connection assignments. The connection assignment symbol can be used to represent multiplexing or renaming. A distinct exchange operator is provided for swapping the values stored in two registers.

DDL

The only data type provided by DDL for the described machine is bits, and integers are provided as compile-time bookkeeping variables. Arrays of bits may be defined with an arbitrary number of dimensions and arbitrary index bounds. Index formation rules are those of ISP (see Sec 2.2.1). Subfields of arrays and collections of registers may be renamed as single arrays. The logical operators AND, NAND, NOR, equivalence, OR, and exclusive or are provided. Addition, subtraction and the usual relational operators are provided along with concatenation, complementation, selective complementation, and reduction for vectors. Arbitrarily complex expressions may be formed. Register and terminal carriers are provided. Connection, transfer, renaming, and bookkeeping assignments are supported.

MOP

MOP assumes that carriers in a machine are composed of bits and it supports any data type encoded in a vector of bits. Vectors and matrices of bits are supported and strong format

specification tools are provided. Rows of matrices may be selected through indexing and fields of a row may be specified. Any expression, including an indexed value may be used as an index. These accessing primitives may be used to define new access modes that can then be used by other parts of the description. Normal programming language data operators are available along with common machine language operations. These operators may be combined to form arbitrarily complex expressions. MOP only allows for register carriers and transfer assignments.

Cassandre

Cassandre supports integers and bits which may be composed into vectors and matrices. Bookkeeping integers are also provided. Constants and ranges of constants may be used as indexes. The result of a decode operator, whose operand is a bit vector, may be used as an index when a data value needs to be used as an index. The following operations are provided: AND, OR, equivalence, exclusive or, reduction of a vector by these first four, concatenation, negation and the previously mentioned decode operator. Arbitrarily complex expressions may be formed using these operators, except the decode operator. If the decode operator is used in an expression, the expression must involve only a direct connection or transfer of the indexed value.

Register and terminal carriers are provided. A connection assignment is provided for terminals and a distinct transfer assignment symbol is provided for transfers. Transfers must be controlled by a clock signal. A bookkeeping assignment is pro-

vided for the bookkeeping integers.

ERES

ERES supports bits as its primitive data type. Bits can be composed into vectors and matrices. If variable expressions are to be used as an index for a row of a matrix, an explicit address register must be specified. Groups of arrays and bits, subsets of arrays, and groups of subsets may be renamed to facilitate multiple views and accessing methods for structures. AND, OR, NOT, addition, and incrementation are provided as primitive operations, as well as others. Any vector may be interpreted as an integer for arithmetic purposes. Arbitrarily complex expressions may be formed.

Terminals and registers are supported as carriers, with connection and transfer assignments being provided. Multiplexing is assumed if multiple connections are specified to the same terminal. Renaming is expressed with an assignment statement.

AHPL

AHPL provides for operations on bits and integers. The integer operations operate on vectors of bits. Support is provided for vectors and matrices of bits with the rows of the matrices being accessed only through constant indices or an explicit indexing operator, while columns may only be accessed through constant indices. It should be noted that bookkeeping integers are provided at compile time which may be used in place of constant

indices. There are no format description tools provided. AHPL provides the standard arithmetic, boolean, relational, and shift operators, as well as absolute value, maximum, minimum, concatenation, decode, encode, reduce and compress operators. A function, syn, is provided for detecting signals from asynchronous systems. It is true if its argument, a signal, was recieved since the last clock signal. Expressions not involving the indexing operator resemble those of APL. The expressions may become arbitrarily complex except that one would want to limit the operators and the complexity of expressions for more detailed descriptions. The indexing operator may only be used where the index is held in a simple register (bit vector) and the indexed value is directly connected or transfered to a terminal or register, respectively.

AHPL provides a variety of carriers and assignments. The carriers include inout terminals, output terminals, registers, busses, ONE SHOTs, and bookkeeping variables. Assignments can be used to describe multiplexed or permanent connection to the terminals and busses. ONE SHOTs have some default value which will change for some period after being set to the non-default value. The delay time (the amount of time the device maintains the non-default value) is specified in the ONE SHOT's declaration. Assignment is used for the transfers to registers and ONE SHOTs, and for setting and changing the value of bookkeeping variables.

Modularization, Control and Data, and Time

CDL

In CDL, one may perform some structural modularization through defining combinatorial networks. Sequences of actions may be defined, facilitating procedural modularization. Both of these definitions are available throughout the description in which they appear. A DO construct is provided to envoke predefined action sequences and an IF..THEN..ELSE is provided for procedural selection (choosing between alternative actions). Activation conditions are used to control actions. All the actions associated with an activation condition are performed in parallel when the condition is true. A single clock may be declared and is required. It is used to describe a machine's behavior with respect to time. Control and data are separately identified with IF..THEN..ELSE and the activation conditions allowing control functions to be affected by data.

DDL

DDL provides for structural modularization through definition of combinatorial networks and declaration of ELEMENTS, special units with unspecified structure and behavior. These special units may be used to suppress detail. Objects declared within a module are only accessible within that module.

Boolean networks are declared with a BO statement. Combinatorial networks (related blocks of terminals and boolean

networks) may be declared using OP statements. An EL statement defines the input and output ports of a component without defining its behavoir or internal structure. The component is available for use within the module where the component is declared.

There are three levels of modules in a DDL description. The top level, called a system, corresponds to the entire described machine. The system is divided into automatons and the automatons are subdivided into segments, with segments being optional. Only the bottommost level may have action statements. Note that this puts a low limit on the amount of nesting possible in a description.

Structural iteration is possible and is aided by a compile time control variable. Procedural constructs analagous to IF..THEN..ELSE and CASE are provided for procedural selection. A macro facility is provided by the ID statement, though parameters are not supported.

Sequencing is controlled through states. Only one state of an automaton may be active at any one time. The actions for a given state are performed in parallel and listed together with a label for the state. A condition may be specified for a state so that if an automaton reaches that state, the actions for the state will not be performed until the condition is satisfied. State changes are explicit operations. The state may be encoded in a memory and the register specifying the current state may be given a name. State changes between segments of an automaton specify a default return state within the current segment. Thus, among other things, an operation may treat a segment as a subprocedure by specifying the current state as the return state. It

is possible to specify a condition for any level module. The actions of that module will be held up until the condition becomes true. It is possible to specify a set of actions for an automaton or segment. These operations will be performed in each state of the automaton or segment, respectively.

Delays and single or multiple clocks may be specified to allow description of the behavior of the described device with respect to time.

MOP

There is only one modularization concept in MOP in the sense of grouping of functions or structures under a name. This concept is the Operand Class, which defines a set of addressing methods for easy reference. Since Operand Classes may overlap, they are a kind of macro facility. However, the language does provide for (force, really) a partitioning of the conceptualization of a computer. The interpretation cycle, memories, data types, addressing modes, operand classes (discussed above), instruction fields, instruction formats and instruction behavior are each treated seperately.

MOP assumes an implicit interpretation cycle which controls which of the described instructions is performed at any given time. Because of this and the fact that MOP decribes very high level behavior, the motivation for control contructs is low. The main control construct is analogous to IF..THEN..ELSE and is used in describing instruction actions.

Behavior with respect to time is described through specifying a time cost for each instruction. Control and data are kept separated except that decisions about actions, including program counter modification, may be based on data values.

Cassandre

Cassandre allows a system to be decomposed into units. Units may be nested and may be connected arbitrarily. This allows the description of any system of interrelationships. One can also define sequences of actions that can be invoked from several places in a unit.

One may specify that multiple units within a given unit may be active. Pulses may be transmitted between units for synchronization and communication. Clocks may also be used for timing and synchronization.

Each action statement is either labeled with a state or is part of a labeled group of statements. Only one state of a unit is active at any one time and transfers between states are explicit. States may be encoded in registers. IF..THEN..ELSE is provided for procedural selection. Structural iteration is provided for parallel execution.

ERES

FRES provides for definition of action sequences and Boolean networks as aids for modularization. Actions and groups of actions are controlled by activiation conditions. IF..THEN..ELSE

is provided for procedural selection.

ERES allows specification of the time required for simple and composite actions. All transfers are dependent on clock pulses. In an extended version, multiple, possibly asynchronous, clocks may be specified and primitives are provided for coordination of asynchronous parts of the system. The basic version of ERES allows for a single clock for the system.

AHPL

The modularization constructs provided by AHPL support the definition of undescribed components and combinatorial networks. These networks are low level modules involving simple data operations and no iteration, though they can describe rather complex patterns of connection. An AHPL statement consists of actions together with branches for deciding the state transistions. The actions can be any mixture of connections and transfers, which are assumed to operate in parallel. The target of a transfer can be made dependent on data and transfers can be conditional. Each statement is numbered and corresponds to a state, and the branches refer to the statement numbers. AHPL provides the APL operators for branching and selection. They provide the normal capabilities, but special characters are used to represent them rather than key words. Structural iteration is provided, including bookkeeping variables. It is possible to specify that a process wait or not wait for the completion of an initiated operation. Control and data are separately identified and selection is provided to allow control decsions to depend on data.

DIVERGE and CONVERGE, which are similar to fork and join, allow for the description of parallel processes. In addition, ONE SHOTs, syn, and delays can all be used to describe timing.

Generality, Readability and Writability

CDL

CDL's main assumption about a device is that it has only one clock. Descriptions in CDL are confined to a primitive register transfer level. CDL can be extended through new operators.

Though comments are supported, CDL's lack of modularization tools greatly restricts readability and writability. Since a description is strictly linear, it cannot follow the structure of the described machine. Also, because the of the global scope of any identifier, large descriptions are going to start having problems with name conflicts, especially if more than one person is working on the same description. It also becomes difficult for a human to spot cooperating parts of the machine and shared facilities. The language is simple, however, and uses familiar notations for standard concepts.

DDL

DDL makes no substansive assumptions about devices being described. Descriptions can be at the register transfer level or switching circuit level. The language is extensible through new

operators and predefined elements (undescribed components).

DDL has been used for design, documentation, simulation and logic design automation. It has been used to describe a variety of machines designed for classroom exercises and research.

The language is rather simple, using many familiar constructs. However, the procedural selection notations are nonstandard and not suggestive. The limited depth of modularization could hamper fidelity and readability. Clusters of closely cooperating automatons are hard to represent and may be hard for the reader to spot. Also, modules that are subcomponents of other modules may be very hard or impossible to represent, again limiting fidelity.

MOP

MOP assumes a device has memories embodying the machine state, instructions to change the state, a main memory to hold the instructions, and an instruction interpretation cycle with a program counter. It is thus limited to describing machines which perform one high level operation at a time in reponse to stored data. Though parallel operations may be used to implement an instruction, they cannot be described with MOP. MOP can be extended through introduction of new constructs to describe the actions of the instructions. MOP is limited to ISP level descriptions. It has been used to describe the PDP8 and PDP11/20 for code generator derivation purposes.

One interesting aspect of MOP is that a MOP description of a computer may be derived, with little human input, from a descrip-

tion written in ISP, which is helpful in light to the impediments to wide applicability of MOP. This method has been used to generate a MOP description of the PDP11/70.

Cassandre

Cassandre assumes that a computer consists a collection of potentially nested automata, each of whose operation is controlled by discrete states with state transitions occuring at clock signals. Actions for a given state are performed in a single clock period. Descriptions are confined to the register transfer level and extensibility is hard to assess from available descriptions (the main language definition is a Ph.D. dissertation written in French). Cassandre has been used to describe a 16-bit ALGOL machine and a system having two linked INTEL 8080's. Automated applications are envisioned which will reduce Cassandre descriptions to lower level descriptions, estimate the cost of units, analyze descriptions, support description modification, do reliability analysis, design circuit test procedures, produce graphical representations of a system, and aid the design of microprogrammed machines.

ERES

ERES as currently defined assumes synchronous systems that are composed of synchronous sequential networks and combinatorial networks. A proposed extension [GJK] provides mechanisms for cooperation between asynchronous parts of a system. The language

can be used on the register transfer level or the switching circuit level, and is extensible through introduction of new operators. Information is not available on what applications have been developed using ERES nor on the machines that have been described in ERES.

The language is syntactically simple and uses familar constructs. The modularization tools seem rather weak, particularly if one wishes to describe a hierarchy. This will hurt both the clarity of descriptions and their fidelity.

AHPL

AHPL can describe register transfer and switching circuit level operations, and provides a completely general parallelism construct. However, the weakness of the modularization constructs will quickly become more and more of a problem as the size of a description grows. This combined with its provision of high-level operators, makes the language mainly useful for behavioral system specifications. The language is simple, but uses special characters which may not be suggestive or familiar to users. Also, because of the problems with modularization cited above, AHPL descriptions will have a hard time retaining fidelity if they go below the level of a specification.

Summary

These non-procedural languages are rather similar to each other, except for the rather distinctive characteristics of MOP. Most of the languages were developed for design, and oriented toward describing the behavior of a system at any level of detail, using RT level primitives. Vectors and matrices of bits are generally supported with vectors of bits being used to hold integers. Except for MOP, no language provides format description tools. Renaming and some restricted form of indexing are usually provided to facilitate accessing parts of composite structures. Most languages provide for arbitrarily complex expressions with addition, subtraction, boolean, relational, concatenation, and complementation operations available for use within expressions. Registers and terminals are generally supported, along with transfer, connection and renaming assignments. Modularization tools are usually limited, with combinatorial networks being provided for structural modularization and short action sequences being definable for procedural modularization. Control constructs are limited, but each provides some facility for alternation. Most languages support clocks as a tool for describing a system's behavior with respect to time. The languages are generally restricted to the RT level or RT and switching circuit levels of description. The languages are usually simple and usually provide familiar facilities, weak modularization tools, and are limited in how closely a description can follow the structure of an implementation.

CDL is distinguished by its generally low level, as seen in its requirement for simple expressions, and is mainly suitable for an implementation level description. DDL, Cassandre, and AHPL are the only languages to support structural iteration and bookkeeping variables. DDL and AHPL are the only languages to provide for use of undefined components and they are also unique in supporting delays. DDL and Cassandre are the only languages to support state encodings. They also provide the more powerful modularization tools. Consistent with its focus on the specification level, AHPL provides a rich array of high level operator and a wide variety of carriers, including some that are rather useful for synchronization with other systems. It is unfortunate that it uses such unfamiliar synactic forms for its constructs. MOP's concern with code generation application appears in many ways, including its high level; lack of constraints on data types, accessing modes and operators; lack of terminals; its form of modularization; and the insistence on time and space requirements for instructions.

Exclusions

This comparison was made as a part of a project investigating compiler retargeting, and therefore mainly examined those languages that would be useful in a retargetable compiler system. Flowware was therfore excluded from the study since it is based on graphic input and the retargetable would require textual input. LOGAL, evidently very useful at Univac for design, was too low level and lacked the extention capability necessary to provide information needed for compilation.

Interesting but Incomplete Languages

This section discusses two languages which are intended for wide utility and ease of use. The languages ConLan and RTS III are not yet complete and therefore cannot yet be exhaustively compared to the other languages in the study.

ConLan

Conlan is being developed by the Conference on Computer Hardware Description Languages and is intended to be suitable for all CHDL applications [Pil77]. The group has been working on this rather ambitious task since September, 1973. Its latest report was distributed in June of 1978. To try to be useful for all applications, Conlan must be able to describe systems on several levels of description and at various levels of detail.

This breadth of levels is being attempted through definition of a primitive, low level language, Primitive Set ConLan, for which some very powerful composition and type definition constructs are defined. These composition and definition tools are then to be used to define more powerful and useful languages. One goal is to standardize the definitions of these more powerful languages in addition to their being defined in terms of the same primitive language.

The ConLan working group has also proposed a two-tiered definition of time with "virtual" time steps within a "real" time step. This will hopefully be sufficient for the needs of any application.

The above concepts are presently only proposals of the Conference Working Group and have not been adopted by the Conference on CHDL's as a whole. Much work still remains in completing the primitive base and the specifying the various application languages based on it.

RTS III

RTS III is being developed by Robert Piloty and his coworkers in Darmstadt [Pi175]. They are focusing on the need for various modularization tools in a CDL that is to be used in a wide variety of applications and in all stages of design and implementation of a system. Current proposals identify externally controlled modules, automatons, open sequences and combinatorial networks. The different designations allow some checking of the body of a module to be sure it has certain formal properties. The constructs are quite general and modules can be nested to any depth. A powerful macro-like facility is defined to facilitate the use of similar components in different parts of a system.

In addition to the segmentation constructs, the basic statement sytax and semantics have been defined. The language is basicly non-procedural, with event conditions controlling groups of actions. There is also an allowance for procedural descriptions. Registers and terminals are supported and can be declared as input, output, or local. In addition, structural iteration and bookkeeping variables are supported.

Conclusions, Remarks

The number and variety of CHDL's proposed and in use is staggering and this study could not hope to be exhaustive. There are at least as many languages and descriptive systems worthy of close study as have been presented here. This study has, however, identified some truly useful languages and some serious flaws in others. Language characteristics that are important to specific applications have been discussed. For instance, MDP is well suited for driving code generation, ISP is by far the most general and flexible language presented, recommending it for a broad range of applications, and SMITE is easy to use and well suited for describing behavior for emulation.

In addition to observations about individual languages, the study has developed and refined a scheme for describing and comparing languages that will be useful for general use. This would be useful in compiling descriptions of a large number of languages for reference. A system of description can also help one see a language more clearly, aiding language design and selection.

Summary of Languages and Characteristics

The following tables summarize the characteristics of the procedural and non-procedural languages separately, with the following remarks applying to both tables. No language allows redundant descriptions unless it is specificly mentioned. All languages that provide shift operators provide for shifts of varying size and in both directions. Any of these languages can be extended by the addition of new data operators, so this is not mentioned under "generality". In all cases the reader should rely on the main text for more detail and precision, these tables are meant as a short overview. The following table defines some of the terms and abreviations used in the summary table.

TERM/ABBREVIATION	DEFINITION
alternation	a construct analogous to IF..THEN..ELSE..
comb. net.	combinatorial network
concat.	concatenation
controlled scope&	ability to control which modules may access a variable
decr.	decrement operator
inc.	increment operator
modular.	modularization
RT	register transfer level of description
selection	a construct analogous to CASE
spec. level	specification level of detail
std. arith.	standard arithmetic operators, at least +, -, *, and /
std. bool.	standard boolean operators, at least AND, OR, and NOT
std. rel.	standard relational operators, at least =, <, >, and their inverses
swit. cir.	switching circuit level of description

PROCEDURAL LANGUAGES

	ISP	APDL	LALSD	SMITE	LCD
PURPOSE	exposition	design, simulation, documentation	design, simulation, documentation	emulation	design
DESC TYPE	behavioral, any detail level	behavioral, spec. level	structural, any detail level	behavioral, any detail level	behavioral, lower detail levels, redundant
DESC LEVEL	RT, ISP, swit. cir.	RT	RT	RT	RT, ISP
DATA PRIMITIVE	bits, integers	binary, octal, decimal, switches	bits, integers	bits, integers	bits, integers, symbolic
COMPOSITION	vectors, matrices	vectors, matrices	vectors, matrices	vectors, matrices	vectors, matrices
FORMAT DESC TOOLS	no	no	no	yes	no
ACCESSING	indexing, renaming	indexing, renaming	explicit address register, renaming	indexing, renaming	indexing

PROCEDURAL LANGUAGES (CONT.)

	ISP	APDL	LALSD	SMITE	LCD
PRIMITIVE OPERATIONS	std. arith., std. bool., std. rel., many shifts, concat.	std. arith., std. bool., std. rel.	inc., decr., std. bool., std. rel., shift, concat., compliment	+, -, std. bool., std. rel., concat.	std. arith., std. bool., reduction by AND
OPERATION COMPOSITION	arbitrary expressions	unspecified	arbitrary expressions	arbitrary expressions	arbitrary expressions
CARRIERS	registers	registers	registers, terminals	registers	registers, terminals, busses
ASSIGNMENTS	transfer, renaming	transfer	connection	transfer	transfer, multiplex

PROCEDURAL LANGUAGES (CONT.)

	ISP	APDL	LALSD	SMITE	LCD
MODULARIZATION comb. net.	procecure, macros	procedures, functions	UNITs, FUNCTIONs, macros	PROCESSes	subprocedure
CONTROL	IF..THEN.. ELSE.., DECODE	IF..THEN.. ELSE.., GOTO, IF EVER, time block	IF..THEN.. ELSE.., CASE, FORK..JOIN, /../ (wait)	IF..THEN.. ELSE.., CASE, DO FOREVER, ESCAPE, PARALLEL-BEGIN & -END	IF..THEN.. ELSE.., WHILE
TREATMENT OF TIME	series-para sequencing, signal-wait	time block	clocks	req. spec., series-para sequencing	rel. timings (implicit clock)
GENERALITY	mult. desc. levels, extension mechanisms	ISP & PT	RT only	RT only	RT only, single clock
READABILITY/ WRITABILITY	familiar, simple, high fidelity	familiar, simple, high fidelity	unfamiliar, complex, controlled scopes	familiar, simple, high fidelity	familiar, simple, high fidelity, redundant desc.'s

NON-PROCEDURAL LANGUAGES

	COL	DDL	MOP	CASSANDRE	ERES	AHPL
PURPOSE	design, simulation	design	code generation	design, development	design	design, instruction
DESC TYPE	behavioral, implementation	behavioral, any detail level	behavioral, spec. level	behavioral, any detail level	structural, any detail level	behavioral, spec. level
DESC LEVEL	RT	RT	ISP	RT	RT, swit. cir.	RT, swit. cir.
DATA PRIMITIVE	bits, integers	bits, integers, state encodings, bookkeeping integers	any bit-encoded data	bits, integers, state encodings, bookkeeping integers	bits, integers	bits, integers, bookkeeping integers
COMPOSITION	vectors & matrices of bits	arbitrarily dimensioned arrays	vectors & matrices of bits	vectors, matrices	vectors & matrices of bits	vectors & matrices of bits
FORMAT DESC TOOLS	no	no	yes	no	no	no
ACCESSING	constant indices, explicit address register, renaming	indexing, renaming	indexing, subfields, user defined mechanisms	indexing operator	constant indices, explicit address register, renaming	constant indices, bookkeeping variable indices, indexing operator in simple expressions

NON-PROCEDURAL LANGUAGES (CONT.)

	CDL	DDL	MOP	CASSANDRE	ERES	AHPL
PRIMITIVE OPERATIONS	+, -, inc., decr., std. rel., shift, std. bool., FETCH, STORE	std. bool., +, -, std. rel., concat., complement, reduction	std. prog. & mach. operations	std. bool., reduction, concat., negation	std. bool. std. arith., and others	std. bool., std. arith., std. rel., abs. value, max., min., shift, concat., reduce, compress, encode, decode, syn
OPERATION COMPOSITION	simple expressions	arbitrary expressions	arbitrary expressions	arbitrary expressions	arbitrary expressions	arbitrary expressions
CARRIERS	registers, terminals	registers, terminals	registers	registers, terminals	registers, terminals	registers, input terminals, output terminals, one shots, busses
ASSIGNMENTS	transfer, connection, renaming, multiplex, exchange	transfer, connection, renaming, bookkeeping	transfer	transfer, connection	transfer, connection, renaming, multiplex	transfer, connection, bookkeeping

NON-PROCEDURAL LANGUAGES (CONT.)

	CDL	DDL	MOP	CASSANDRE	ERES	AHPL
MODULARIZATION	comb. net., action sequences	comb. net., elements, automatons, segments, macros, subprocedure	op. classes, M-ops, access modes	action sequences, units	bool. net., action sequences	comb. net., LABELs
CONTROL	IF..THEN.. ELSE.., DO (like CALL), activation conditions	alternation, selection, structural iteration, wait conditions	alternation	IF..THEN.. ELSE.., structural iteration	IF..THEN.. ELSE.., activation conditions	selection, ->(goto), ;(parallel transfers), structural iteration, DIVERGE, CONVERGE
TREATMENT OF TIME	single clock	clocks, delays	req. spec.	clocks, pulses	clocks, req. spec.	delays, one-shots
GENERALITY	RT only, single clock	RT or swit. cir. desc. level	ISP only, inter. cycle assumed	RT only, various applications	RT or swit. cir. desc. level	RT or swit. cir. desc. level, LABELs, synch. primitives
READABILITY/ WRITABILITY	weak modular., simple, familiar, low fidelity	weak modular., familiar, low fidelity	simple, familiar	simple, familiar	weak modular., simple, familiar, low fidelity	simple, unfamiliar, weak modular., low fidelity

Appendix B:

A Comparative Study of Intermediate Languages

An Intermediate Language (IL) represents a program while it is being processed by a compiler, and allows communication of information about the program between phases of the compiler. Whether a compiler consists of one or of several physical passes, it can be thought of as a collection of several logical phases. The first phase (ordinarily lexical analysis) inputs a source program and outputs some intermediate language program. Other phases will accept an IL program and output another, possibly in a different IL. Eventually some phase accepts an IL and outputs the target language. The form of these ILs, and the information carried along with them (symbol tables, data flow information, etc.) depend on the functions of the phases that they link. In designing a compiler, it is often convenient to use several ILs, and to let their operators and storage mechanisms reflect the source language, or the target language, or both. ILs may be dependent on the language structure, or on the host machine or both. These dependencies make it difficult to retarget or adapt the compiler.

The use of a single, universal intermediate language has been proposed to reduce compiler implementation effort by providing a standard interface between the language dependent and machine dependent parts of a compiler [Co174]. The three major advantages of a compiler built this way are: the compiler can be adapted to another language by changing only the language dependent phases, it can be retargeted by rewriting the machine-dependent phases, and a considerable part of the compiler (some optimizations, symbol table routines, etc.) will be relatively constant. The major requirements on a universal IL are that it should be independent of both the source language and the target machine, and that it be flexible enough to represent all the information that has to be communicated between phases of the compiler.

IL Requirements

The first major requirement mentioned in the previous section is that a universal IL be source language and target machine independent. One result of this requirement is that the "level", or complexity of the operators and storage descriptions, of the language be well placed between the source

languages and target languages that we are considering. Two observations motivate this requirement. The first is that if the mismatch between the IL and either the source or target languages is too great, the corresponding translation will be difficult, and the second is that if the IL is too close to either language, it will be dependent on it to some extent. It is difficult, if not impossible, to define a fixed, universal, intermediate language [HW78]. Take for example, an implementation of FORTRAN on the PDP 11/780 VAX machine. The high level DO construct should remain a DO in the IL, since 1) knowledge of high level control constructs aids global optimization and, 2) the VAX has a single instruction which implements almost the entire control of a DO loop. However, handling such a peculiar control construct (from a modern language viewpoint) makes the IL rather language dependent. Most ILs would have it reduced to separate loop and test elements, which, of course, are both hard on the global optimization and make the use of the DO instruction on the VAX very hard to implement. For these reasons, a universal IL must be extendible. Extendibility will partially negate some benefits of having a single fixed IL, but it will make the universal IL idea workable.

Another requirement for an intermediate language is that it be able to represent all the information that has to be passed between the compiler phases that it links. A single IL is desirable so that only a single support system to read, write, and/or analyze it will be needed, and because a single, convenient IL can provide a conceptual framework for the compiler. Therefore, a universal IL should be able to represent not only the semantics of the language, but also the information required for register allocation, code generation and optimization. It is also desirable that the language be suited to the transformations required for optimization.

The level of an IL results from the selection of its operators and control constructs, and from its storage mechanisms. Since there is more commonality among the source languages we are considering than among the target machines, a fairly high level representation for operators and control constructs is needed. Also, the retention of looping and conditional constructs

in the source language allows the compiler to retain information useful for optimization. As for storage mechanisms, it is important to avoid the use of specific accumulator registers or stack operations that are not machine independent.

Languages Considered

The intermediate languages considered in this report, with a brief description of each follow.

The JOCIT IL [Dun75]

The JOCIT IL was developed for compilers translating JOVIAL/J3 for a number of machines. The language is high level and fairly language dependent. It is a post-fix - polish language. Looping operators are included, and symbols and their attributes are kept in a symbol table.

OCODE [Ric71]

OCODE is the IL for Richards' BCPL compiler. It is language dependent, has been used to retarget the compiler to from 1020 machines (as of 1974) and is not well suited for stack machines.

HALMAT [II74]

HALMAT is the IL for the Intermetrics HAL/S compilers. It is high-level and language dependent. It uses explicit temporaries for intermediate results, and is machine independent.

JANUS [Col74, HW78, WH78, CPW74]

JANUS was designed as a universal IL. It provides a large set of operators and a flexible storage scheme, and it is extendible. Its control and data structures are at a fairly low level, and it uses a stack for intermediate results. It is designed so that it can be translated to assembly code by a macro processor, and some difficulty in handling temporary variables and its lack of high level control constructs can be attributed to this.

$TCOL_{ada}$ [Cat78, SLN79]

TCOL was designed by R. G. G. Cattell as a universal IL. The version presented in his thesis is sketchy, and $TCOL_{Ada}$ is an elaboration on it developed by the PQCC project at Carnegie-Mellon University. TCOL programs are trees, with fairly high level data structuring and control flow operators. PQCC's approach to universality is to introduce language or machine dependent operators and data structures as needed ($TCOL_{Ada}$ is the version tailored to Ada) and to let the compiler-compiler produce a compiler tailored to that version of TCOL.

IL Comparison

The table in this section gives the ILs mentioned above, and how well each meets the requirements that have been defined. The text in this section will elaborate on the entries in the table.

1. Source and Machine Independence

Since OCODE, HALMAT and the JOCIT IL were each designed with a particular language in mind, they tend to contain assumptions about the source languages operators, data types, parameter passing conventions, etc. and so their language independence is low. JANUS was designed for language independence, and its use in Pascal, Algol 68 and BCPL compilers confirm that this goal was well met [HW79]. TCOL was also designed to be language independent, and $TCOL_{Ada}$ was designed as an IL for Ada. Much of the development of $TCOL_{Ada}$ was in the specification of data types, data structuring facilities, control structures, etc. Since Ada has a rich set of these features, and since $TCOL_{Ada}$ is at a much lower level than Ada, $TCOL_{Ada}$ is fairly language independent. The structure of the IL and its external representation and symbol handling are language independent.

The machine independence of all these languages is good, and all of them except TCOL have been used in compilers for more than one machine.

2. Level

The three language-dependent ILs are at a fairly high level, with looping, alternative and procedure call constructs. JANUS has a mix of high and low level features. Its control constructs are conditional jumps, its data structuring mechanisms are less complex than those of TCOL but more complex than those of most ILs, and its procedure call and parameter passing mechanisms are very flexible. TCOL is the highest-level of the ILs considered. Its control constructs, data types, data structuring methods, and operations have more flavor of Ada than of a machine language.

	Jocit IL	OCODE	Janus	$TCOL_{Ada}$	HALMAT
Source Independence	Poor	Poor	Good	Fair	Poor
Machine Independence	Good	Good	Good	Good	Good
Level	High	High	Medium	High	High
Temporary Storage	Implicit Stack	Stack	Stack	Tree	Explicit Temps
Extendibility	No	No	Yes	Yes	No
Suitability for Code Generation	Fair	Fair	Fair	Good	Fair
Practical Experience	Yes	Yes	Yes	No	Yes

Figure B.1 IL Comparison Chart

3. Temporary Storage

HALMAT uses explicit temporaries while the Jocit IL is a post-fix polish language and so uses an implicit stack. Janus and OCODE use explicit stacks and TCOL is a tree language, with temporary storage implicit in the tree structure.

4. Extendibility

With enough effort any language is extendible, however, Janus and TCOL were designed with extendibility in mind, while the other ILs were not.

5. Suitability for Code Generation

This is discussed in the IL selection in Section 3 of this report.

6. Practical Experience

HALMAT and the Jocit IL have each been used in commercial compilers for more than one computer. OCODE has been used to transport the BCPL compiler to a number of machines, and Janus has been used in several compilers and an experimental number of machines. $TCOL_{Ada}$ has not been used in a compiler to date, but will probably be used for one or more Ada compilers in the near future.

Appendix C:

A MOP Description Example

```
{MΘP file for Mini-S (simplified PDP-8)}          Note: Comments are in
                                                        braces {}.

{I-flds}[
(ΘP 0 3 0 Θ)
(I.BIT 3 1 0 C)
(ADR 4 8 0 Θ)
(IΘ.BITS 4 8 0 Θ)
(UBITS 5 7 0 Θ)
(UCLASS 4 1 0 Θ) ]

{SBs}[
(PC 1 8 P)
(Mp 256 12 M)
(Acc 1 12 G)
(IΘ.REG 1 8 R)
(L 1 8 R) ]

{AMs} [
%8:      $1:#8
%Mp:     (<> Mp $1:#8 0 12)
%@Mp:    (<> Mp (<> Mp $1:8 0 12) 0 12)
%PC:     (<> PC 1 0 8)
%Acc:    (<> Acc 1 0 12)
%L:      (<> L 1 0 8)
%IΘ.REG?(<> IΘ.REG 1 0 8) ]

{ΘCs} [

Y:  (
%8 :: (EMIT[5 0 0]  $1 0)
%Mp ::  (EMIT[5 1 0]  $1 1) )

Z: (
%Mp :: (EMIT[5 1 0] $1 0)                    from R. G. G. Cattell [Cat78]
%@Mp ::  (EMIT[5 2 0] $1 1) )
```

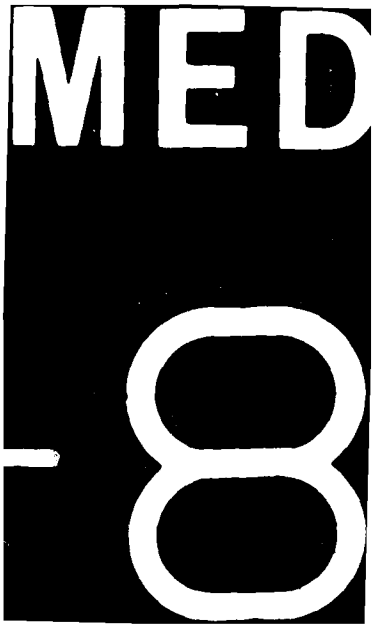
```
IΘ: (
%8 :: (EMIT[6 0 0]  $1) ) ]

[
{I-FMTs}
{FMT 1} (ΘP Z)              {1-opnd format}
{FMT 2} (ΘP Y)              {jump format}
{FMT 3} (ΘP UCLASS UBITS)   {micro format}
{FMT 4} (ΘP IΘ)             {IΘT format}

{ΘC-FMTs}
{FMT 5} (ADR I.BIT)         {Y and Z}
{FMT 6} (IΘ.BITS) ]         {IΘ}

{Mops}[

(← %Acc  (AND  %Acc $1:Z)) ::
(EMIT[AND  1  1 1]  0 $1)

(← %Acc (+ %Acc $1:Z)) ::
(EMIT[TAD 1 1 1]  1 $1)

(; (← $1:Z  (+ $1:Z 1))  (=> (EQL $1:Z -1) (← $PC (+ %PC 1)))) ::
(EMIT[ISZ  1 1 1]  2 $1)

(; (← $1:Z %Acc) (← %Acc 0)) ::
(EMIT[DCA 1 1 1]  3 $1)

(; (← %L %PC) (← %PC $1:Y)) ::
(EMIT[JMS 2 1 1]  4 $1)

(← %PC $1:Y) ::
(EMIT[JMP 2 1 1]  5 $1)
```

```
(← %IO.REG IO) ::
(EMIT[ IOT 4 1 1]  6 $1)

(← %Acc (NOT %Acc)) ::
(EMIT[ COMA 3 1 1]  7 0 40)

(← %Acc 0) ::
(EMIT[ CLRA 3 1 1]  7 0 20)

(← %Acc (+ %Acc 1)) ::
(EMIT[ INCA 3 1 1]  7 0 10)

(← %Acc (- %Acc 1)) ::
(EMIT[ DECA 3 1 1]  7 0 4)

(← %Acc (↑ %Acc 1)) ::
(EMIT SLA[ 3 1 1]  7 0 1)

(NO.OP) ::
(EMIT[ NOP 3 1 1]  7 0 0)

(← %Acc 1) ::
(EMIT( SET1A 3 1 1]  7 0 30)

(← %Acc 2) ::
(EMIT[ SET2A 3 1 1]  7 0 31)

(← %PC %L) ::
(EMIT[ RTS 3 1 1]  7 1 40)

(← %PC %Acc) ::
(EMIT[ JMPA 3 1 1]  7 1 20)
```

```
(=> (LSS %Acc 0 ) (← %PC (+ %PC 1))) ::
(EMIT[ SKPL 3 1 1]  7 1 4)

(=> (EQL %Acc 0)  (← %PC (+ %PC 1))) ::
(EMIT[ SKPE 3 1 1]  7 1 5)

(=> (NEQ %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[ SKPNE 3 1 1]  7 1 2)

(=> (GTR %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[ SKPG 3 1 1]  7 1 1)

(=> (LEQ %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[SKPLE 3 1 1]  7 1 6)

(=> (GEQ %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[ SKPGE 3 1 1]  7 1 3)
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


MED
8